A variational approach to mapping: an exploration of map representation for SLAM

by

Saad Rustam Khattak

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of

Master of Science

in

The Faculty of Science
Computer Science

University of Ontario Institute of Technology
July 2012

Acknowledgments

Fresh out of high school, I opted to go to U of T and studied life sciences for two years before realizing it was not my calling. My aptitude has always been towards computer sciences and game development. Without the strong support from my father (Lt. Col. (R) Abdul Hameed), and my mother's (Rashida Neelam) constant prayers, I would have never been able to make the transition to University of Ontario Institute of Technology (UOIT) and certainly would not be where I am today. My parents have always had very high expectations of me and I hope that I have met and exceeded them.

At UOIT I was under the mentorship of several professors who catapulted me to where I am today. During my undergraduate years from 2006-2010, I was guided by Dr. Bill Kapralos, Dr. Khalil-el-Khitab and last but certainly not least, Dr. Andrew Hogue. They have been instrumental to my growth as a student and eventually taken the big step towards graduate studies.

I started my graduate career under the guidance and mentorship of my current supervisor, Dr. Andrew Hogue. I quickly learned that if one hopes to be honest to a research project and really wants to contribute to the scientific world then following a beaten track is not an option. Suffice it to say, without Dr. Hogue's guidance and tremendous help, this thesis would not see the light of day. I have tremendous respect for him as a supervisor, teacher and a friend. His knowledge in and out of his area of expertise is humbling and I feel indebted to him for giving me an insight and peaking my interest in the field of computer graphics and robotics.

I would like to thank OGS for providing the funding during my graduate studies. A special thanks goes to my colleagues (Daniel Buckstein, Man Kang and Brent Cowan) who have offered their support and helped me in areas where my shortcomings were obvious. Finally, I would like to show my gratitude and acknowledge

my Alma mater, University of Ontario Institute of Technology which provided me with an environment and resources to carry out and continue my research.

I hope that this thesis is the highlight of my graduate studies and comes up to the expectations of all those who reposed confidence in me and humbly desire that it contribute in its very small capacity to the scientific research in the field of computer science and robotics.

Abstract

Simultaneous Localization and Mapping (SLAM) algorithms are used by autonomous robots to build or update maps of an environment while maintaining their position simultaneously. A fundamental open problem in SLAM is the effective representation of the map in unknown, ambiguous, complex, dynamic environments. Representing such environments in a suitable manner is a complex task. Existing approaches to SLAM use map representations that store individual features (range measurements, image patches, or higher level semantic features) and their locations in the environment. The choice of how the map is represented produces limitations which in many ways are unfavourable for application in real-world scenarios. In this thesis, a new approach to SLAM is explored that redefines sensing and robot motion as acts of deformation of a differentiable surface. Distance fields and level set methods are utilized to define a parallel to the components of the SLAM estimation process and an algorithm is developed and demonstrated. The variational framework developed is capable of representing complex dynamic scenes and spatially varying uncertainty for sensor and robot models.

Keywords: SLAM, Level Set, Distance Fields, Implicit Surfaces, Variational, Mapping, Deformable Model

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Simultaneous Localization and Mapping (SLAM)

Many robotics applications require or mandate little to no human intervention. For example, robots programmed for space exploration, subsea mining and tunnel exploration. These *autonomous* robots must be able to map an unknown or outdated environment in order to estimate their pose thus making localization and mapping the core problem [1]. Localization and mapping is a hard problem to solve because of factors such as inaccurate and unreliable sensor data and dynamic unpredictable environments [1].

Generally, a robot depends on a predefined map and estimates its location within this map. Maps can be inaccurate, incomplete or simply unavailable and thus the robot must map the environment and localize itself simultaneously [1]. Algorithms that accomplish this task are known as Simultaneous Localization and Mapping (SLAM) algorithms. SLAM algorithms for mobile robotics attempt to solve two problems simultaneously:

1. Mapping: determining what an unknown environment looks like.
2. Localization: determining where the robot vehicle is located within this map.

1.2 Uncertainty

If perfect data could be obtained from sensors then a perfect map of the environment can be generated with accurate localization of the robot. Unfortunately,

even very high quality sensors are unable to provide error free data which adds uncertainty to the system.

Autonomous robots need to accommodate uncertainty to be able to traverse a dynamic environment. A dynamic environment adds to uncertainty, however, it is not the only source of uncertainty the robot must be able to deal with. Uncertainty also arises with erroneous data collected through a myriad of imperfect sensors, referred to as sensor noise, which a robot must use to perceive the environment. Furthermore, actuators (Motors) that drive a robot can be unpredictable which increases uncertainty. The severity of unpredictability depends on the quality of the actuators. Various factors such as wear and tear, control noise and mechanical failure degrade the quality of even the highest quality actuators resulting in an increase in uncertainty [1].

Computers add to the level of uncertainty simply because they can only represent numbers within a certain level of precision. Therefore, all internal models of the environment that can be represented on the software level are approximate and will accumulate error and are known as model errors [1].

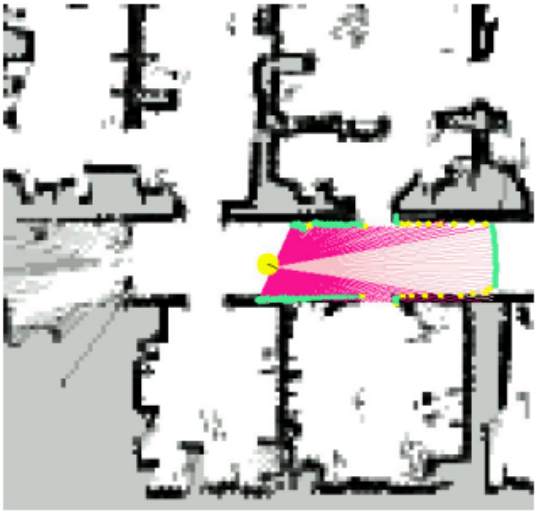
Autonomous robots are real-time systems, and as such must perform calculations that can be performed within a small slice of time. This limits the amount of computation that can be performed thus limiting accuracy for the sake of speed and thereby increasing uncertainty.

Working with erroneous data described above requires modelling the uncertainty to be able to represent ambiguity mathematically. For example, in Figure 1.1(a) the robot generates the likelihood of different positions from a laser scan (Figure 1.1(b)). Robots that work under tight constraints, such as industrial assembly robots, reduce the level of uncertainty they need to deal with, thus reducing complexity of the algorithms that drive the robots. Most autonomous robots however, are expected to cope with a large degree of uncertainty due to the factors discussed earlier. Many methods have been proposed to model the uncertainty and are in use in modern mapping algorithms. Collectively, they fall under the category of probabilistic algorithms [2].

1.3 Problem Statement

Probabilistic robotics is the dominant approach in the field of robotics mapping [2]. Probabilistic algorithms do not rely on a single guess for decision making

(a) Laser scan and part of the map



(b) Likelihood for different positions

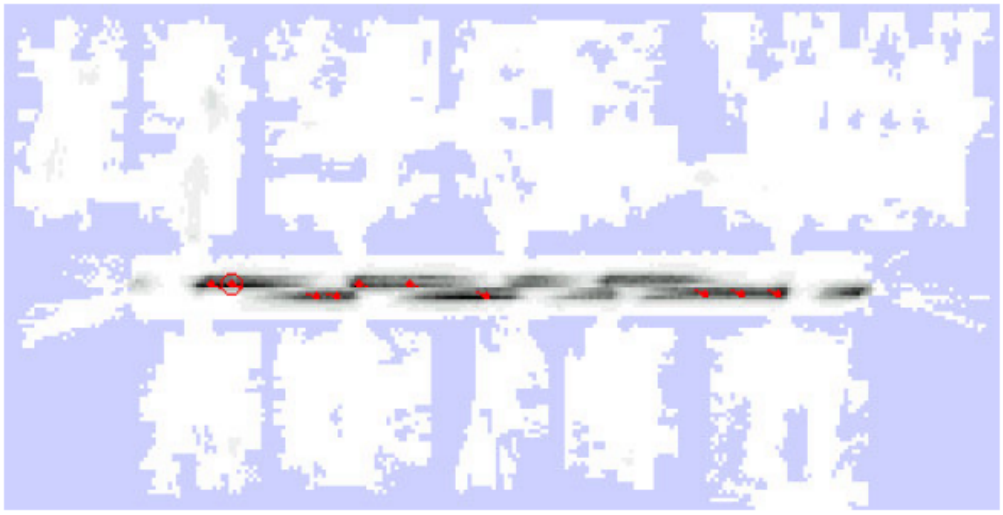


Figure 1.1: Robot position likelihood [1]

but base the decisions on a complete set of guesses through the use of probabilistic distributions. This allows the algorithms to be robust and forgiving when assessing an environment and are able to deal with the ambiguity inevitably arising in unstructured environments.

Navigation in probabilistic robotics requires knowledge about the surrounding environment as well as knowledge pertaining to the robot's current location in the environment. To be able to gather data from the surrounding environment, an autonomous robot must perceive the world through sensing where typically range sensors are used.

Estimation of an autonomous robot's current location relative to an external reference frame is known as robot localization [1]. Localization requires data about the environment, which can range from complete maps of the environment to placement of known beacons. The accuracy of localization depends directly on the accuracy of the predefined map or beacon locations. The initial position of the robot can also be predetermined, however, that is of little use in real world scenarios. Thus, current research in robot localization focuses on algorithms that allow a robot to localize itself in an unknown environment.

Most SLAM algorithms rely on feature-based maps. Each feature (range measurements, image patches, or higher level semantic features) is a unique landmark in the environment that is tracked and updated whenever it is encountered. Such feature-based maps are effective in specific scenarios (e.g. approximately planar motion, static environments) and are most successful when the environment can be satisfactorily described by a small set of discrete features. However, such representations limit the application of SLAM in large-scale real-world dynamic environments.

Feature based maps are found to be inadequate for representing large and complex environments because of the sparsity of the features [3]. The system's complexity is increased exponentially with each new additional feature. Managing a very large set of features (continually increasing in size) becomes challenging even for modern computing hardware. Furthermore, dynamic environments permit improper data association of measurements (because of shape deforming aquatic life) leading to algorithm divergence. Any reasoning about the environment between discrete features is difficult without heuristics and the estimation of unconstrained motion in 3D space is challenging.

Current SLAM algorithms have not been shown to work well in complex 3D scenes with unconstrained sensor motion nor do they work well in dynamic environments. Underwater environments filled with fish and shape-deforming aquatic life is one such example [4]. Such an environment poses a significant hurdle for existing SLAM algorithms.

Instead of representing the world with explicit salient features, the map can be represented as an implicit function which is the focus of this thesis. Thus, SLAM can be redefined in a coherent manner and gives the opportunity to map differentiable and dynamic processes that current mapping techniques fail to handle.

1.4 Contribution

This thesis explores the use of varational techniques (methods that attempt to minimize or maximize the energy in a system) in the context of SLAM. The developed techniques and frameworks use distance fields to represent the environment and level set methods (methods for the manipulation of implicit surfaces) as the fundamental mathematical framework to re-formulate SLAM as a contour/surface tracking problem.

Later chapters will present level set frameworks capable of generating, displaying and manipulating implicit surfaces in real-time via the level set methods. The framework addresses disadvantages of implicit surface generation and manipulation by using spatial data structures and scalable multi-threaded memory managers. The resulting framework is shown to be fast, efficient and scalable.

1.5 Thesis Structure

Chapter 2 will discuss related work done in SLAM and the level set methods and proposes to bridge the gap between the two different fields of research.

Chapter 3 will discuss the theory behind implicit surfaces, level set methods and distances. The chapter discusses in detail the concept of signed distance fields and manipulation of the implicit surface through the signed distance fields. Advantages and disadvantages of the level set methods are highlighted as well.

Chapter 4 discusses a novel approach to mapping an environment for use in SLAM algorithms. The notion of using sensors as surface deformers is discussed and the theoretical implementation of the proposed technique is provided.

Chapter 5 discusses in detail the software frameworks and the methods used to allow real-time surface manipulation even on large grids.

Chapter 6 shows the application of the frameworks and the resulting maps generated by it.

Finally, in Chapter 7 the thesis is concluded with a summary and discussion of the work, its limitations and future work.

Chapter 2

Related Work

2.1 SLAM

Autonomous robots are required in a host of real-world applications requiring little to no human intervention. They are used domestically (e.g. autonomous vacuum cleaners), industrial (e.g. factory manufacturing robots), and military (e.g. unmanned drones and missiles). These autonomous robots depend on complex algorithms to guide them in potentially unknown and possibly dynamic environments. For the robot to traverse an environment predictably algorithms must be developed to map the environment and the robot must calculate its position and orientation relative to this map. Navigation of dynamic environments, by its nature, requires the ability to accommodate uncertainty because of unreliable sensor data, physical limitations such as wheel slippage and dynamic environments (environments with moving objects).

Even with *a priori* knowledge of a map (obtained through a blueprint of an environment) an autonomous robot must modify and update the map as the environment changes. In a controlled environment this may not be necessary, but a dynamic environment demands that this particular ability be present in all autonomous robots. Subsequent modification of the map is important as usually a given map is not an accurate representation of the environment or the map may be outdated. In some scenarios, an initial map may not even exist (unknown regions such as other planets). Thus, for robots to be truly autonomous, they must be able to create and update maps and compensate for changes in the environment that are not reflected on the map automatically.

Localization of a robot requires a predetermined map of the environment while map generation requires knowledge about a robot's local position in the environment. Simultaneous localization and mapping (SLAM) algorithms attempt to solve both problems, where there is incomplete knowledge of the environment and the *pose* of the robot itself. The goal is to estimate the joint probability over robot pose and a particular map representation [1, 5].

Smith, Self, and Cheeseman [5] introduced the idea that inaccuracies in measurements of observed features in an environment could be reduced while simultaneously reducing the error in the overall trajectory by formulating the problem in a Bayesian framework to estimate relationships between robot pose and observed features. Csorba [6] observed that with time, all features become fully correlated with each other and with the robot location - a key result that defines the need for probabilistic map representation for SLAM.

Maps

Most SLAM algorithms rely on simple to use feature-based maps where each feature is a unique landmark in the environment. The sparsity of the feature based maps makes them inadequate for representing complex environments. Tracking more features increases the complexity of the system exponentially [1]. Effective demonstration of SLAM in real-world systems are usually based on metric feature based maps, constrained sensor motion (planar, 3-DOF[1]) and use sensors that provide range data (e.g. laser line scanners and stereo cameras) [7].

Current feature-based methods have not successfully used SLAM in a complex 3D scene with unconstrained motion (6-DOF) [8] because of complications arising from sensor motion in large, dense, dynamic/deformable environments. One such example is a dynamic underwater environment filled with fish and shape-deforming aquatic life such as sea weed and coral reefs. These environments pose significant hurdles for existing SLAM algorithms.

Maps created through the SLAM process have found uses in path-planning to determine and predict object trajectories and avoid obstacles. Navigation limited to one plane is usually represented by a 2D top-down map to characterize spatial occupancy. With sensors capable of 6-DOF motion, mapping is closely related to 3D modeling.

[1] Degrees of Freedom

Current SLAM algorithms make extensive use of state estimation and sensor fusion algorithms such as Kalman Filters, Particle Filters and Expectation Maximization [1]. In the following sections, existing mapping techniques and the supporting algorithms are reviewed to see how they represent maps and the advantages and disadvantages of each implementation is discussed. The notion of implicit surface representation in the context of SLAM is explored since it has the potential to represent complex, dynamic scenes. Although there are many more variations, the most common techniques will be discussed.

2.1.1 Kalman Filters and Feature Based Maps

Feature based SLAM algorithms extract sparse features from the sensors information. A 3D feature is represented as an $\Theta_i = [x, y, z]^T$ point in three-dimensional space. The set of features $\Theta_t = \{\Theta_1, \Theta_2, \Theta_3, \cdots, \Theta_n\}$ are used in conjunction with the robots pose, s_t, to build the final SLAM state that is estimated as $X_t = (s_t, \Theta_1, \Theta_2, ...\Theta_n)$. The map is comprised only of features resulting in more memory and computational efficiency by sacrificing the quality of the map as much of the information from the sensors is discarded [9].

Kalman filters [10] are recursive state estimation algorithms that represent posterior probability density functions using Gaussian distributions [11]. A posterior probability is the probability of an event after relevant data is processed. Gaussian distributions are frequently used because of the small number of parameters required for complete representation [2]. There are variations to the Kalman filters (which require linear dynamic systems), extended Kalman filter (which can handle nonlinear dynamics) and the extended information filter (a reformulation of the EKF using information matrices rather than covariances). Many current approaches to SLAM algorithms use the EK filters [12, 13, 14] as well as the Extended Information (EI) filters [15, 16].

EKF SLAM

Maps generated by EKF SLAM algorithms are feature based. The *features* represent the landmarks that the map is composed of which the robot is able to recognize and use as reference points. Landmarks are especially useful for correcting accumulating (but consistent) errors using a technique known as loop closing (Figure-2.1).

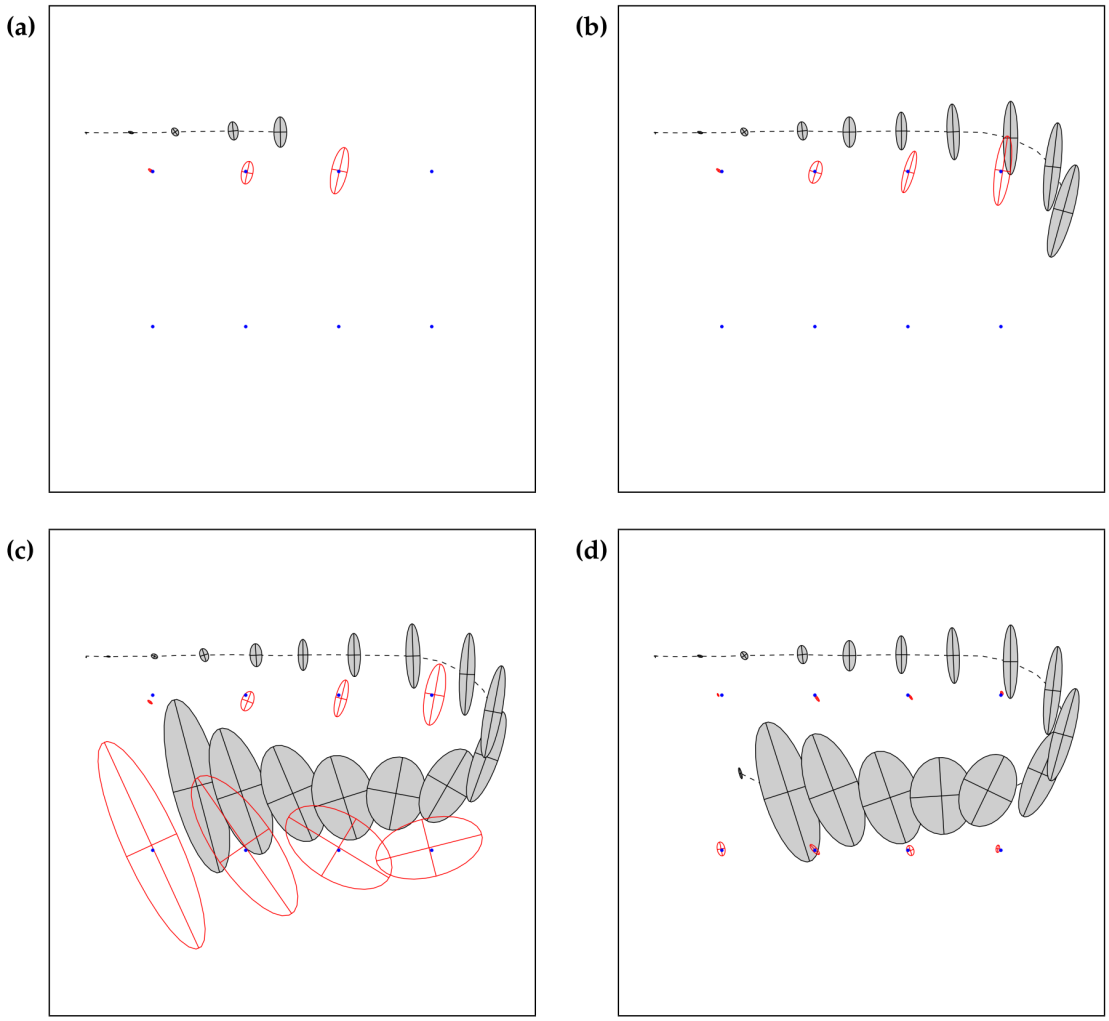


Figure 2.1: A robot's path is a dotted line and its estimates of its own position are shaded ellipses. Eight distinguishable landmarks of unknown location are shown as small dots, and their location estimates are shown as white ellipses. (a) The robot continues along a straight path with low uncertainty of the current pose (b) The robot's pose changes and the uncertainty of the pose with relation to the landmarks rises (c) The uncertainty continues to rise as the robot's pose continues to change (d) The robot sense the first landmark again and the uncertainty of all landmarks decreases as does the uncertainty of the current pose [1].

In most EKF SLAM implementations, a state vector X of the system at time t is comprised of the current pose estimate as well as landmark estimates

$$X_t = [s_t, \Theta_1, \Theta_2, ...\Theta_n] \tag{2.1}$$

where s_t is the pose at time t and Θ_n are the landmarks. The complexity of the algorithm is $O(n^2)$ where n is the number of landmarks because of a necessary matrix inversion of an $n \times n$ matrix.

Smith et al. [17] proposed solving the SLAM problem using the EKF. The paper correlates a robot's pose errors with errors on the map via a covariance matrix. EKF SLAM algorithms solve the online SLAM problem (estimates most recent pose and map) and by nature are unable to solve the full SLAM problem (generate map using all available information) stated as

$$p(x_t, m|z_{1:t}, u_{1:t}) \tag{2.2}$$

Where x_t is the pose at time t, Θ is the map and $z_{1:t}$ and $u_{1:t}$ are the measurements and controls, respectively [1].

The process of relating sensor measurements to estimate landmark (features) is called correspondence. A landmark encountered by the robot that is not recognized but used in computations has unknown correspondence [1] and should be either added to the list of features or discarded because of insufficient saliency.

EKF SLAM algorithms have major drawbacks in terms of increase in computational complexity when there is an increase in the number of landmarks [18, 19]. Sensor updates require quadratic time in the number of landmarks to compute because the covariance matrix increases in size with the addition of each landmark. The matrix also needs to be inverted to estimate the Kalman gain weight factor after each measurement update. This complexity puts an upper limit on the number of landmarks that can be used by the algorithm on a given computational processor and thus reduces its scalability and effectiveness in large, complex environments.

Guivant et al. [20] proposed compression filters to demonstrate that the complexity of SLAM algorithms due to sensor updates can be reduced, although it increases uncertainty. Leonard et al. [21] proposed a derived algorithm, called De-

coupled Stochastic Mapping (DCM) algorithm, which divides the environment into multiple globally-referenced submap regions thereby reducing the computational requirements on a per region basis. Both algorithms have limitations (limited number of landmarks and scalability issues with complex environments) when it comes to approximations of the environment in an attempt to reduce computational requirements and the resulting increase in uncertainty.

Additional limitations to the EKF SLAM algorithm become apparent when the algorithm is applied with unknown correspondence of landmarks. The algorithm works well only when the landmarks are distinguishable within a certain error. This is because the algorithm applies an incremental maximum likelihood estimator to the correspondence problem [1]. The algorithm therefore performs poorly when landmarks are highly ambiguous in which case multiple, distinct maps are obtained of the same region from different data sets. Because of the incremental nature of the algorithm, it is impossible to take into account all incoming sensor data for posterior probability computations.

While the EKF SLAM algorithm is proactive and the other algorithms (e.g. GraphSLAM) simply accumulates information and processes it offline. Sparse Extended Information Filter (SEIF) algorithms combines both methods into one, albeit with some approximations of sensor data. The feature relations of the map are represented sparsely in the case of the covariance matrix. Since landmark information is modified by local neighbours the measurement update can be performed in constant time O(1). Although, in order to perform reasoning about world references or obtain metric features, the information matrix must be inverted which results in quadratic complexity. SEIF integrates the past robot poses similar to that of the EKF SLAM algorithms while at the same time maintaining information representing all the knowledge of the paths and poses similar to the GraphSLAM technique described below.

GraphSLAM

The EKF SLAM algorithm has quadratic computational update complexity in measurement updates which makes it inefficient to track a little more than a hundred landmarks [1]. Under specific formulations, the full SLAM problem has been shown to form sparse graphs of the posterior [22] and is the sum of non-linear quadratic constraints.

EKF SLAM represents information via a covariance matrix along with an average vector of the robot's path while GraphSLAM represents the information as a graph of *soft constraints* [23, 22]. Since GraphSLAM works on a sparse tree of *soft constraints*, it requires an additional pass over the collected information to build the map and the robot path. This is called the lazy SLAM technique. In contrast, the EKF SLAM algorithm(s) maintain the best estimate of the path as well as the map at all times. Thus the extra phase of computation, not present in EKF SLAM, must be taken into account with GraphSLAM algorithms and variations of it.

GraphSLAM does not need to resolve the information that is accumulated over time and thus no calculations need to be performed during that stage. This allows GraphSLAM algorithms to acquire maps that are much larger than maps that can be generated by the EKF SLAM algorithms. Having access to the full data when constructing a map, GraphSLAM algorithms can apply linearization and data association techniques not possible with EKF SLAM where previous data is discarded once it has been integrated. GraphSLAM can revise previous data associations and can perform multiple passes of the data if necessary, which results in superior and more accurate maps that those generated by current EKF SLAM algorithms.

The added benefits come with a cost. More specifically, GraphSLAM algorithms work on a fixed size data set whereas EKF SLAM can update the map indefinitely (of course, taking into account the large computation and memory requirements of the covariance matrix, this advantage is short lived). The size of the sparse graph grows linearly with time, and as such would require ample amounts of memory to store. The EKF algorithms have no such constraint. Lastly, complex sparse graphs will require offline calculations for map generation and as such, GraphSLAM algorithms cannot be used where the robot must create, modify and update the map indefinitely.

FastSLAM

FastSLAM is an algorithm that combines EKF and particle filtering to solve the SLAM problem. It was first introduced by Montemerlo et al. [18] to address shortcomings in the EKF SLAM algorithms noted previously. FastSLAM is computationally more efficient than EKF only SLAM algorithms in that it can be implemented in a time logarithmic vs. time quadratic in the number of landmarks

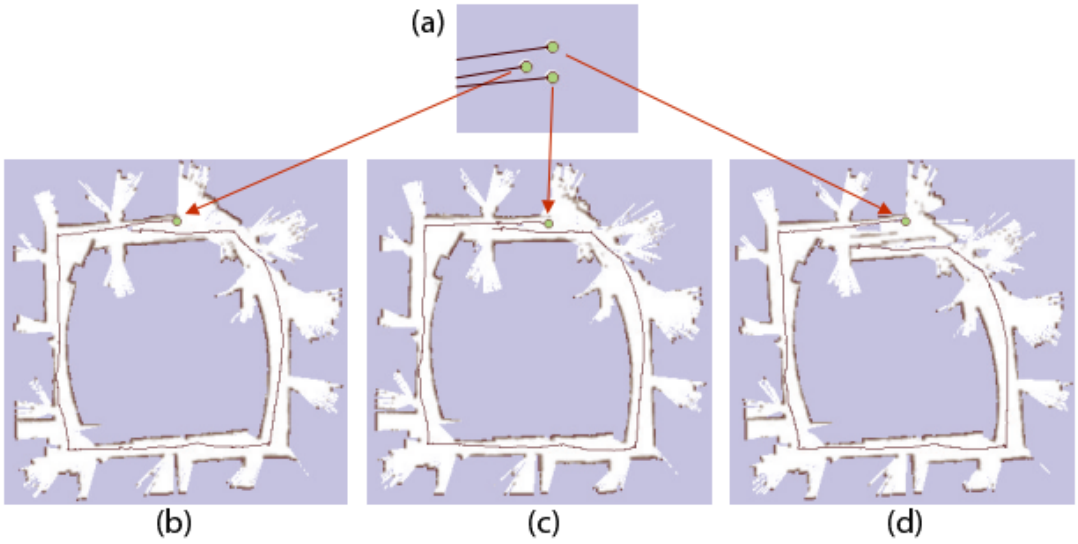


Figure 2.2: The FastSLAM algorithm (in this case a grid-based variant shown) application (a) Three particles (green dots) are shown where each particle carries its own map shown in (b, c, d) [24]. Generally, more particles are used when generating feature based maps using the FastSLAM algorithm.

[18]. It is able to achieve the efficiency by keeping the individual landmark measurements independent of each other given the knowledge of the robot's path.

FastSLAM uses a particle filter to estimate the posterior. The algorithm maintains a set of particles, where each particle denotes a unique landmark, which represent the posterior and corresponds to the complete path of the robot within some uncertainty. The map errors and features are defined as independent EKFs. The features are represented in a similar manner when compared to EKF SLAM but instead of estimating a large and growing state in the Kalman filter, N separate and smaller Kalman filters are instantiated; one per feature. The resulting covariance matrix is 2×2 for 2D features and 3×3 for 3D features, which is very fast to invert, resulting in a complexity of $O(MlogN)$ where M is the number of particles and N is the number of landmarks [19].

FastSLAM is able to deal with the previously mentioned data association problem, i.e. discerning ambiguous landmarks, resulting in the algorithm being more robust in unknown correspondence. Another advantage of FastSLAM algorithms over the previously discussed EKF SLAM algorithms is the fact that particle filters can accommodate non-linear robot motion models, where the EKF algorithms model the motion through linear or close-to-linear functions.

Algorithms that improve upon FastSLAM have been suggested by Nieto et al. [25] and Montemerlo et al. [26]. The former extends the algorithm by addressing the data association problem using a nearest neighbour technique. Their paper also describes a new implementation to handle uncertainty in the data association problem called Multiple Hypothesis Tracking. The latter paper aims to overcome some deficiencies in the FastSLAM algorithm.

2.1.2 Occupancy Grid Map Representations

The occupancy grid approach was first proposed by Elfes et al. [27] where they gathered measurements from sonar units to generate occupancy information that was projected onto a two-dimensional map (Figure 2.3). The technique combined several readings from individual sensors, with some constraints, to reduce the uncertainty which was then represented as probabilities in a discretized grid. The range readings provided information about each grid cell as occupied or empty. The proposed technique then used probability density functions projected on a horizontal plane to generate a map of the environment.

A range measurement from sonar does not contain enough information to make an accurate assumption of the occupancy of a particular grid cell. Elfes et al. [27] combined a number of readings thereby reducing the uncertainty of a given region and used a recursive Bayesian update to estimate the probability of a cell being occupied where a Bayesian update or Bayes filter estimates, recursively, a probability density function using a set of measurements. The algorithms for relating two maps of the same area in the proposed technique assume the environment is composed of straight lined objects. This is due to the resolution of the occupancy grid. A high resolution grid is required to represent curved objects. Dynamic objects pose a significant problem for this particular technique as there is no separation between the robot's local area and the entire environment. Detecting objects smaller than the cell size is also problematic (e.g. furniture legs).

Crowley et al. [29] address the above mentioned drawbacks including modelling dynamic obstacles and accurate position estimation of the robot. They proposed using two occupancy grids where the first grid models within a close vicinity of the robot and the second grid models the environment. The *local* occupancy grid can then be used for collision detection with a fine enough resolution to detect smaller objects. This method also relies on the environment and obstacles to be composed of straight lines to calculate probability of occupation which limits its use in environments with non-linear obstacles.

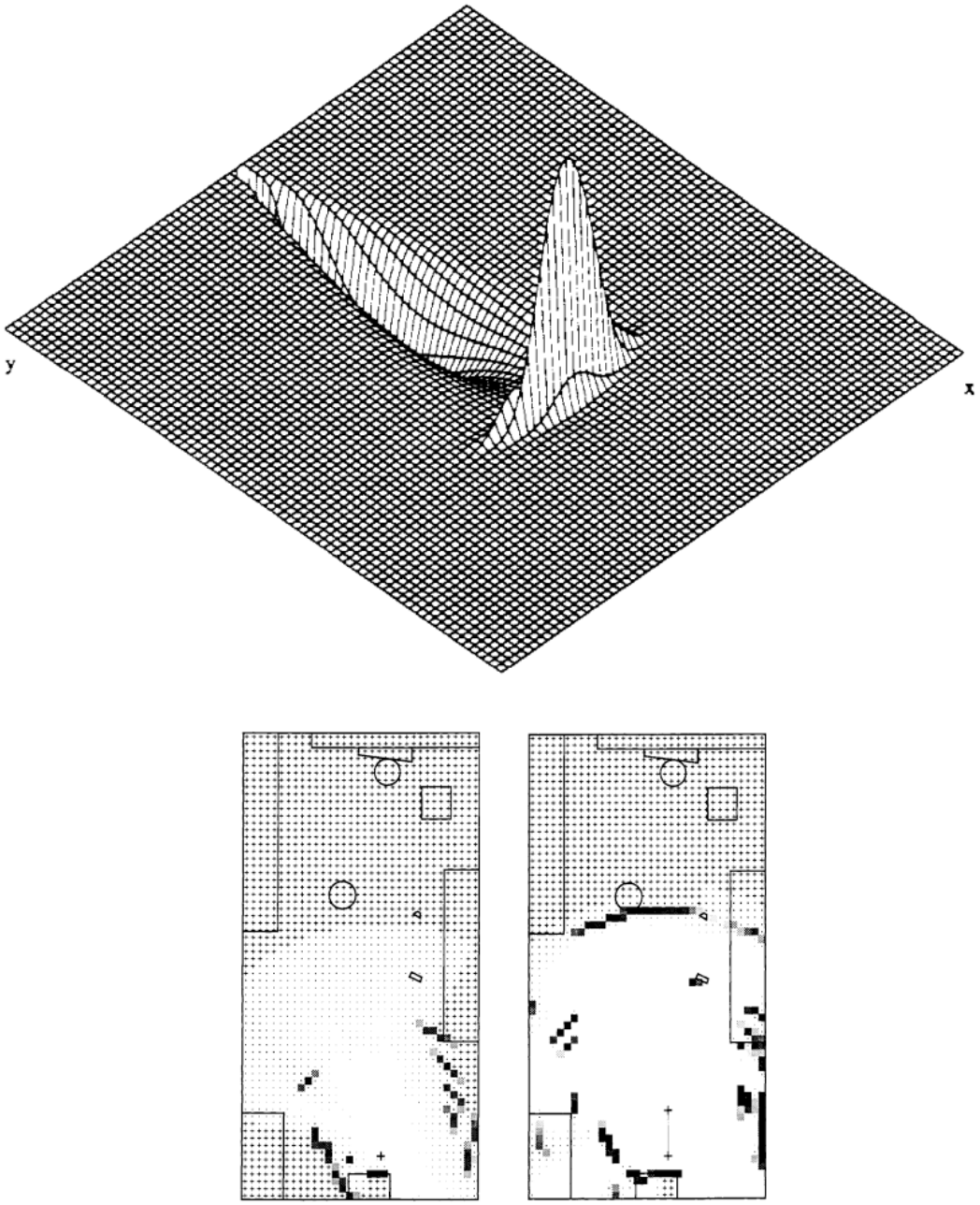


Figure 2.3: top: Sonar sensor model showing uncertainty, bottom: occupancy grid generated by sonar [28].

2.1.3 Volumetric and Mesh based Map Representation

All the mapping techniques mentioned in the paper so far operate in 2D. This works well as the robot itself is confined to a plane. Volumetric representations of the environment extend the map into the 3^{rd} dimension thereby packing much more information than the 2D counterparts. The extra information can be used to differentiate between different environmental features while also facilitating merging of same features [29].

Thrun et al. [24] successfully demonstrated a real-time algorithm that can generate large volumetric maps of a cyclic environment (Figure 2.4). The proposed algorithm uses incremental mapping and posterior estimation allowing fast mapping of indoor environments.

Thrun et al. [30] proposed an algorithm to generate volumetric 3D models of the environment from range and camera data. The algorithm uses a variation of expectation maximization to fit 3D data to simple planar models. It takes advantage of the assumption that the environment consists of flat surfaces allowing the final map to be less complex with reduced noise. The proposed algorithm relies on differentiating objects in the environment to track changes. Without the *markers* the algorithm may fail to disambiguate two separate pieces of a planar wall.

The previous technique failed to extend into the vertical dimension which Triebel et al. [31] address by using two forward pointed laser scanners. Each sensor returns measurements in the opposite plane to accommodate for uneven terrain. The volumetric map of an underground mine is constructed using sensors pointing perpendicular to the robot's heading direction. The system does fail with highly uneven terrain and other natural obstacles (e.g. mud holes) found in mines.

Although the above algorithms were able to generate volumetric maps which were then converted to mesh based representations, there were some assumptions and limitations. The environment was assumed to consist of planar (or relatively planar) structures and did not contain obstacles. None of the mentioned techniques were demonstrated to cope with moving obstacles. This can be attributed to the computationally expensive collision detection of the resulting parametric surfaces. Mesh based representations of the map required post-processing which was not demonstrated to be performed in real-time.

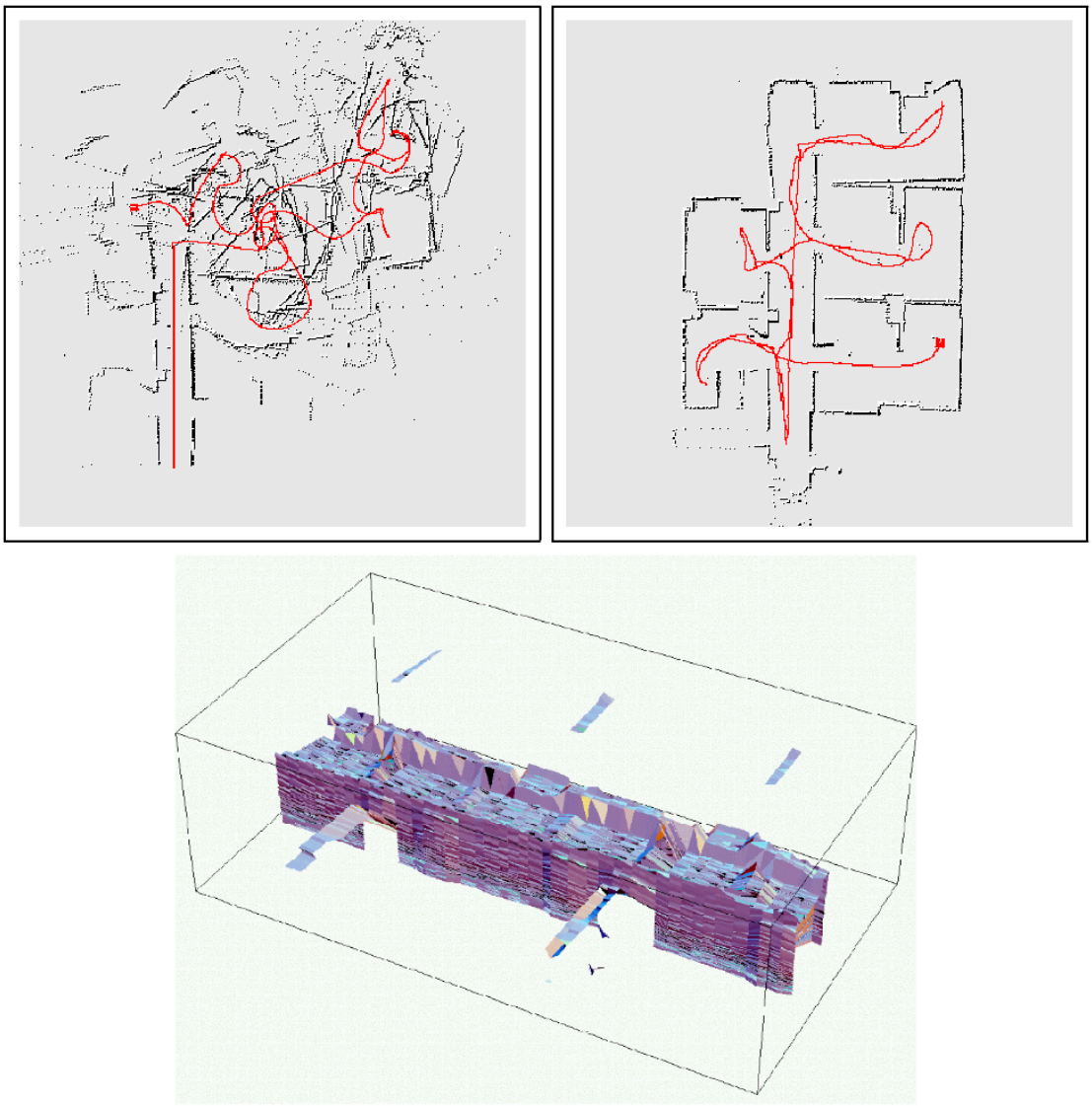


Figure 2.4: Top: Autonomous exploration and mapping using an urban robot: Raw data and final map, generated in real-time during exploration, Bottom: 3D map of a corridor [24].

2.2 Implicit Surfaces and Mapping

Implicit surfaces, represented by implicit functions [32, 33], level sets [34, 35, 36, 37] and distance fields [38, 39] constitute variational approaches to 3D object modeling. Their use has not yet been explored in the context of probabilistic map representation for SLAM. The methods have proven effective for 3D modeling and tracking changes in topology of 3D models with relative ease.

All of the approaches of map representation discussed so far use discrete structures or parametric surfaces. Such representations are easy to generate (i.e. computationally inexpensive) and memory requirements have no relation to the size/volume of the model, but rather the surface's complexity.

The term *variational method* refers to an estimation algorithm utilizing an energy-based model and seeks to find a stable state that minimizes (or maximizes) explicit constraints [40]. Nonlinear partial differential equations (PDEs) are used to apply these constraints on the evolution of the energy model to locate local (or global) minima. Level set methods are one example of a variational method enabling the efficient tracking of an implicit-surface (or contour) that minimizes specified constraints.

Level set methods (LSMs) have proven to be useful for representing active contours (i.e. contours that change shape over time [40]) regardless of the surface complexity. Generally, LSM maintains a contour not as a set of curve elements (or features) but rather implicitly as a spatially continuous function representing the distance to the contour. The contour itself is extracted, as needed, as the zero-level set of this function [41].

PDEs are used to evolve the energy function and model non-linear dynamics. Implicitly, the evolution of the energy model changes the shape of the contour allowing the representation of complex deformable shapes. Multiple objects can be tracked by extracting the *zero* level set of a function containing multiple peaks/valleys, resulting in multiple contours. This has previously been exploited successfully for image segmentation [37, 42]. Despite the significant strengths of the method, the only notable use of LSM within the SLAM literature is the research by Guivant et al. [43].

In [43], the authors used LSM to evaluate an entropy function establishing the quality/amount of information to be gained by a particular trajectory in the motion

planning stage. However, their algorithm relies on a combination of topological and feature-based maps to represent the world. The resulting sparse map, although usable for navigation, is unusable for 3D modeling or other quantitative analysis.

Variational methods provide efficient mechanisms to evolve surfaces (contours) based on a set of forces or constraints. This enables highly complex shapes and surfaces to evolve from very simple initial conditions. For example, LSMs can be used to represent evolving 3D models such as fluids. Applying fluid dynamic forces to the LSM function allows for the surface to deform and act as a fluid [44].

2.2.1 Level Sets

Sensor data is inherently noisy and has insufficient resolution to represent a complex real world environment with a single set of measurements [1]. Reconstruction of surfaces with current mapping techniques is one of the most challenging tasks to undertake and the quality of the reconstruction is limited by the method employed to represent the surface. Navigation algorithms which use the resulting maps increase in complexity as the complexity of the environment increases negatively affecting the robot's path planning solutions.

Implicit surfaces are defined by a function that equals some constant, such that the function satisfies all the points representing the surface. The function itself is defined such that for a point x the function $F(x) > 0$ if x is outside the surface, $F(x) < 0$ if x is below or inside the surface and $F(x) = 0$ when x is on the surface (Figure 2.5). A continuous range of values can be generated from the function called a signed distance field. Continuous distance fields must be discretized into spatial grids if they are to be represented by computers. The discretized distance fields, although continuous, are an approximation of the original signed distance field. A more detailed analysis and theory is provided in Chapter 3.

Surface Reconstruction

Robust reconstruction of 3D objects is a very powerful feature of implicit surfaces as demonstrated by [12, 45]. Smooth surfaces can be generated from segmented data while preserving fine detail. Errors in geometry, such as holes, can be filled resulting in a continuous surface.

Carr et al. [45] uses Radial Basis Functions (RBFs) for reconstructing geometry from point-cloud data. RBFs are functions where the values depend only on the

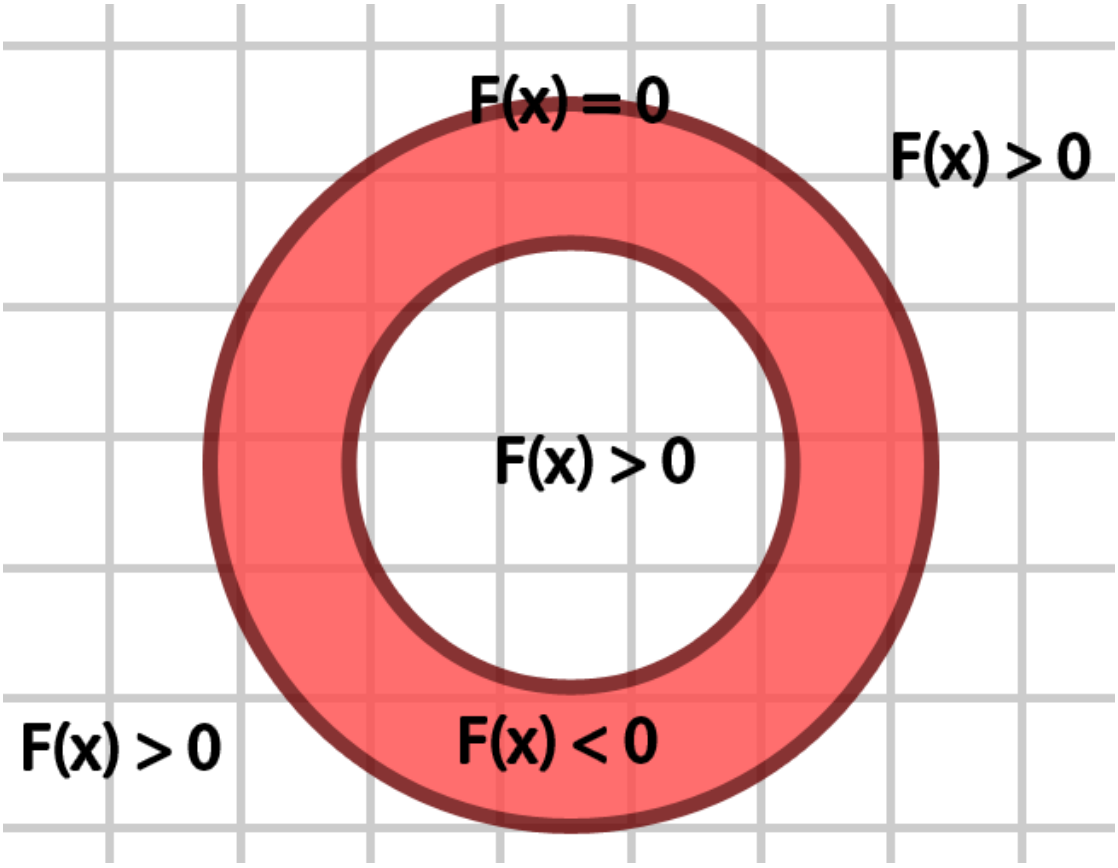


Figure 2.5: The dark red outlines represent the surface itself, where the level set function $F(x) = 0$. The lighter red *interior* of the surface is where $F(x) < 0$. $F(x) > 0$ in all regions that are exterior to the surface. This figure shows a 2D slice of a 3D level set function of a torus.

distance from the origin. The technique describes generating valid normal vector data by off-setting the points. Various sensors in robotics, such as laser sensors and depth sensors generate point cloud data which can then be used in conjunction with RBFs to reconstruct the original surface.

RBFs are too slow to be used for online SLAM problems. Ohtake et al. [46] propose a multi-scale approach to data reconstruction, where they first use spatial down sampling to construct coarse-to-fine hierarchy of point sets and then interpolate the sets of starting from the coarsest level. This method is insensitive to the density of scattered data and allows it to be faster and more robust when filling large holes in a surface or range data.

Another method proposed by David et al. [12] utilizes the signed distance field and fill the gaps by extending the signed distance function which is defined only near the holes of the surface. This is achieved by diffusing the value of the distance field near the hole into undefined areas allowing the zero level set of the distance field to propagate and fill the hole. This method is an iterative process and requires several iterations. The paper also proposes an extension to the algorithm called space carving allowing correct topology to be generated in close spaces. The technique has a limitation where ripples may appear in areas where the laser scans are not perpendicular to the surface. Even so, the surface reconstruction is able to provide a good approximation to the original surface.

Surface Propagation

Surface propagation in distance fields is a very stable operation where propagation (if performed with a proper Δt) results in a continuous surface regardless of the change in surface volume. Parametric surfaces (e.g. polygonal meshes) can stretch and break if manipulated too much without further subdivisions.

Interpolation with parametric surfaces is difficult or impossible with complex concave geometry [47]. Current methods produce many artifacts and incorrect surface representations in different slices. Morphing two implicit functions on the other hand is an easy operation as it only requires the interpolation between two distance fields (given some weighting) resulting in an accurately morphed curve or surface.

Distance fields have been used for morphological operations of erosion and dilation. A morphological operation of erosion removes external parts of a surface while dilation will add parts to the boundary of the object. Holes in a surface can increase

or decrease in size depending on the direction of interpolation [48]. Morphological operations have also been carried out on binary segmented data, such as an MRI[2] scan, by Hastreiter et al. [49] which can be easily translated to data collected by robotic sensors.

Path Planning and Collision Detection

Point clouds, by nature, cannot be used for collision detection unless a surface has been constructed using the point data. Collision detection between parametric surfaces has been an active area of research and is used in many real time applications and games, although there are many constraints that are put in the system to ensure proper collisions. Parametric meshes used for collisions are usually very simple, convex and are a crude approximation to a detailed surface. This is because of the complexity involved with determining point to plane and plane to plane intersections.

Collision detection with distance fields is much simpler and faster. A point in the distance field returns a negative distance signifying the point is inside the surface, a positive distance signifying the point is outside the surface and a value of zero to signify the point is on the surface. Since implicit surfaces are continuous, to perform calculations on the surface, the surface must be discretized and this results in approximations. Thus, all collisions done on and/or between implicit surfaces are approximate where the resolution of the collisions is determined by the number of voxels in the distance field grid that is approximating the surfaces.

Teschner et al. [50] use spatially partitioned distance fields for collision detection of deformable bodies . They proposed methods for accurate and fast intersection checks for complex deformable bodies such as cloth and skin. Data structures for distance field collision detection require a larger memory footprint than their parametric counterparts. The authors propose a solution based on adaptively sampled distance fields to reduce the memory footprint, although that will increase the computation time to generate the distance fields of the concerned bodies.

Fast collision detection makes it possible to efficiently find paths which may result in computational improvements in existing algorithms. Path planning and collision detection are important facets of autonomous robots where distance fields may prove to be a superior choice.

[2]Magnetic Resonance Imaging

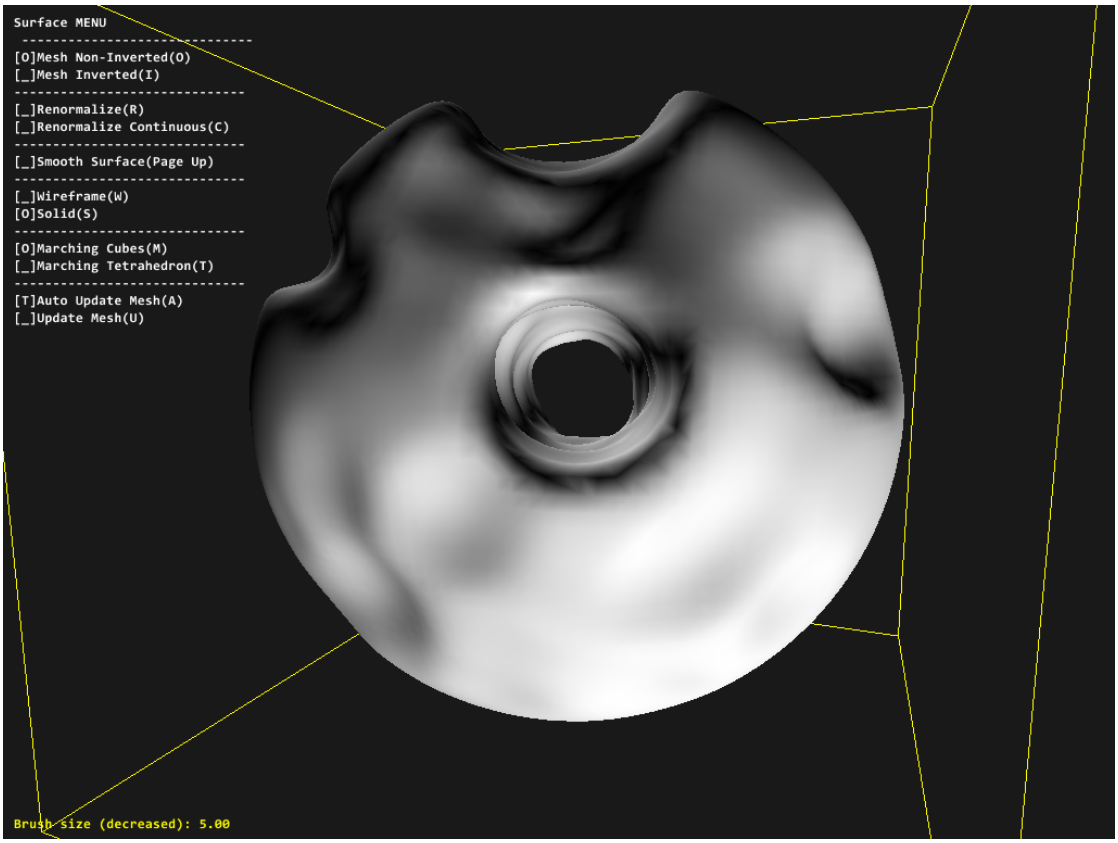


Figure 2.6: A sculpt using the LSM framework and the default sculpting tools. In this case the initial surface was an implicit sphere. The center of the object has a *hole* punched through and through which is very difficult to achieve in traditional sculpting applications that use polygons.

Surface Manipulation

One widely used application of distance field methods is digital sculpting used in animation and game production (Figure 2.6). Adding detail to a continuous surface in an interactive fashion is a difficult task. Increased detail means finer voxel grids which translate to greater memory requirements and greater computational costs for creating, manipulating and rendering the surface. Various methods have been proposed to allow interactive editing and manipulation of these surfaces.

Museth et al. [51] present a set of operators for manipulating level set surfaces. The deformation of the implicit surfaces is done via a speed function, a collection of which is described in the paper. Five editing operators were shown, including blending, smoothing and sharpening. The method works by applying velocity to the implicit surface in the direction of the surface via an operator with a center of interest. The operations are guaranteed to produce non-self-intersecting, closed surfaces.

2.3 Level Set Methods and SLAM

Applying LSM to SLAM requires reconciliation between distance-field representations (for LSM) and probabilistic spatial representations (for SLAM). This is a challenging task requiring efficient ways to represent uncertainty models for measurements, the vehicle motion, and the entire state (which includes trajectory and map). SLAM's power stems from its ability to reduce the uncertainty of the state through sensing and data association. Within an LSM-based approach, forces that distort the surface towards areas of higher certainty can model the uncertainty per measurement. Integrating multiple measurements requires an efficient way to estimate the spatial influence of the combined set of measurements.

In SLAM, the sensing process can be redefined as surface deformation. Range measurements obtained, for example from laser scanners or stereo cameras, can be used as *tools* to modify an initial (inaccurate) representation of the environment. The data from the sensors will *sculpt* the initial surface as the robot senses the environment. For a detailed description of the proposed method, see Chapter 4.

Chapter 3

Representing Surfaces

3.1 Introduction

A geometric surface can be represented by a few different mathematical concepts ranging from polygonal isometric surfaces (and variations of it) to mathematical representation of the surface, such as non-uniform rational b-splines (NURBS) [52] which create continuous surface patches that can then be combined to represent a complex surface.

Most graphics applications today render using parametric surfaces, with the most basic element being a triangle with three vertices. Modern Graphics Processing Units (GPUs) specialize in rendering, manipulating and texturing millions of triangles per frame. Parametric surfaces however have drawbacks that do not make them an ideal choice for some applications. For example, deformation of a complex parametric surface is tricky and morphing between two topologically different parametric surfaces is very difficult or impossible.

In contrast to parametric surfaces, implicit surfaces (described in detail in the next section) are able to address the above mentioned drawbacks along with having some other major advantages. For example, fast collision detection on complex, concave surfaces, which is very computationally expensive on parametric surfaces. Implicit surfaces have their own drawbacks and thus do not have the same hardware support as parametric surfaces. The drawbacks include high *memory usage* and *computational requirements*.

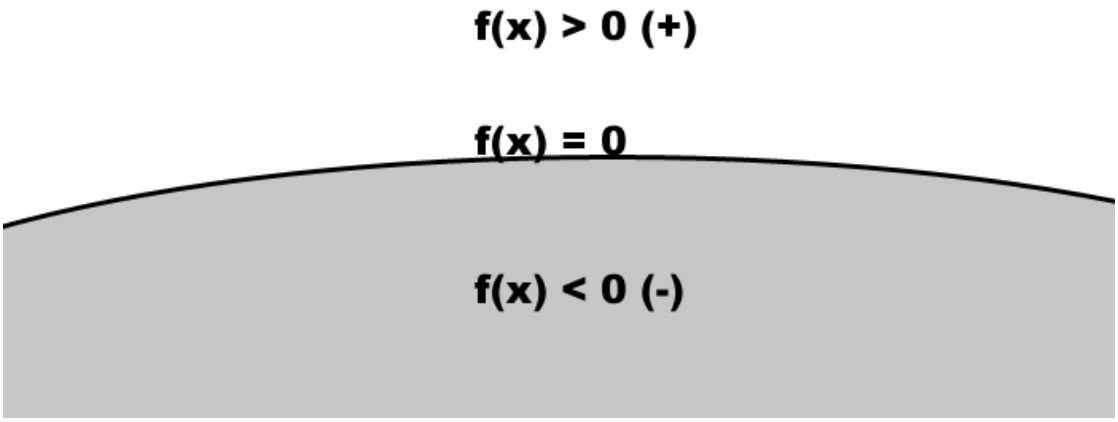


Figure 3.1: An implicit surface where the black border signifies the surface boundary. The grey area is the inside of the surface and white the outside. The implicit function $f(x)$ that defines the surface returns a signed distance to the surface for any given point. The sign, combined with the distance returned by the function, is useful for determining whether the point lies outside, inside or on the surface.

In this chapter the theory behind implicit surfaces will be discussed including representation of implicit surfaces via distance fields.

3.2 Implicit Surfaces

An implicit surface is defined by a continuous function that equals some constant such that the function satisfies all the points representing the surface [41]. If a point lies on the surface, the function returns zero. Thus, it can be easily determined whether an arbitrary point in space lies exactly on the surface or not. The function is also able to determine, using the same concept, whether a point lies *outside* or *inside* the surface depending on the *sign* returned by the function (Figure 3.1).

3.2.1 Signed Distance Fields

Implicit surfaces are represented by continuous functions and by nature cannot be represented by discrete data types. The surface must therefore be approximated by discrete sampling. The higher the number of samples, the more accurate the final approximation of the original implicit surface (Figure 3.2). Generally, the discretization encompasses the area or volume (depending on whether the surface representation is 2D or 3D) of the surface and is represented by a grid of cells or *voxels* (volume elements).

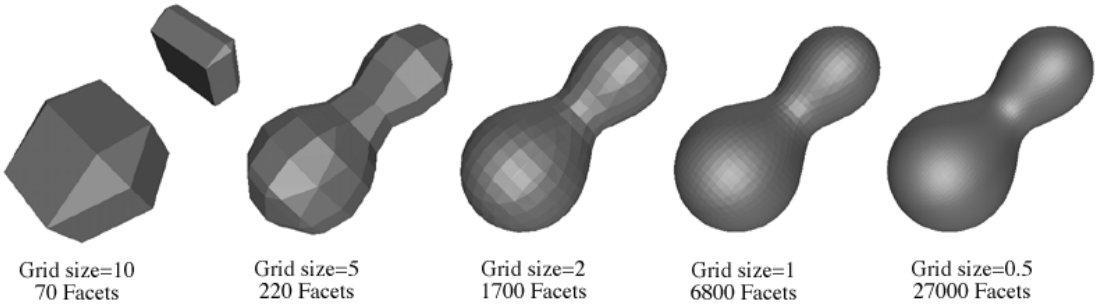


Figure 3.2: An implicit surface sampled with varying grid sizes.

Each sample stored in a voxel represents the distance to the surface being represented. When an implicit surface is first constructed by discrete sampling, the distance is usually given by querying an implicit function that defines the desired surface. In addition to the distance information, the function returns a *sign* which allows one to determine whether the sample lies inside (negative) or outside (positive) the surface (a sample lies on the surface if the distance is zero). The *signed distance* information is then stored in voxels which, collectively, make up the *signed distance field*.

Even with very high resolution grids, few (if any) voxels will contain a zero distance from the surface. Thus, the surface is found by detecting a *change in sign* across neighboring voxels. In some applications, such as medical imagery from an MRI scan, only the signs are stored as binary data where a single bit may represent + or - sign depending on its state. The change in sign is then used to determine where the surface intersects.

In Figure 3.3, each cell or voxel stores a signed distance to the surface which is the shortest distance to the surface. It is not immediately apparent where the surface contour lies, especially since none of the voxels have a zero distance. However, the voxels where the surface might be intersecting can be established. This can be done by simply looking at neighboring voxels. If the neighboring voxel undergoes a sign change, it can be deduced that the surface lies somewhere between the voxels undergoing a sign change. Figure 3.4 shows the same grid but with the surface (in this case a circle with a radius of 1.5) visualized.

With 3D distance fields, the surface can be visualized using polygonal meshes. Similar to the 2D grid example, intersecting voxels can then be polygonized (Section 3.2.2) to obtain a coarse approximation of the original implicit surface. A naïve approach to compensate for the course result and to obtain a better approximation is to increase the grid's resolution (i.e. increase the number of samples).

0.62	0.08	0.08	0.62
0.08	-0.79	-0.79	0.08
0.08	-0.79	-0.79	0.08
0.62	0.08	0.08	0.62

Figure 3.3: A 2D 4x4 signed distance field grid approximating a surface. Note that the values in this field are not arbitrary but represent an actual surface which can be seen in Figure 3.4

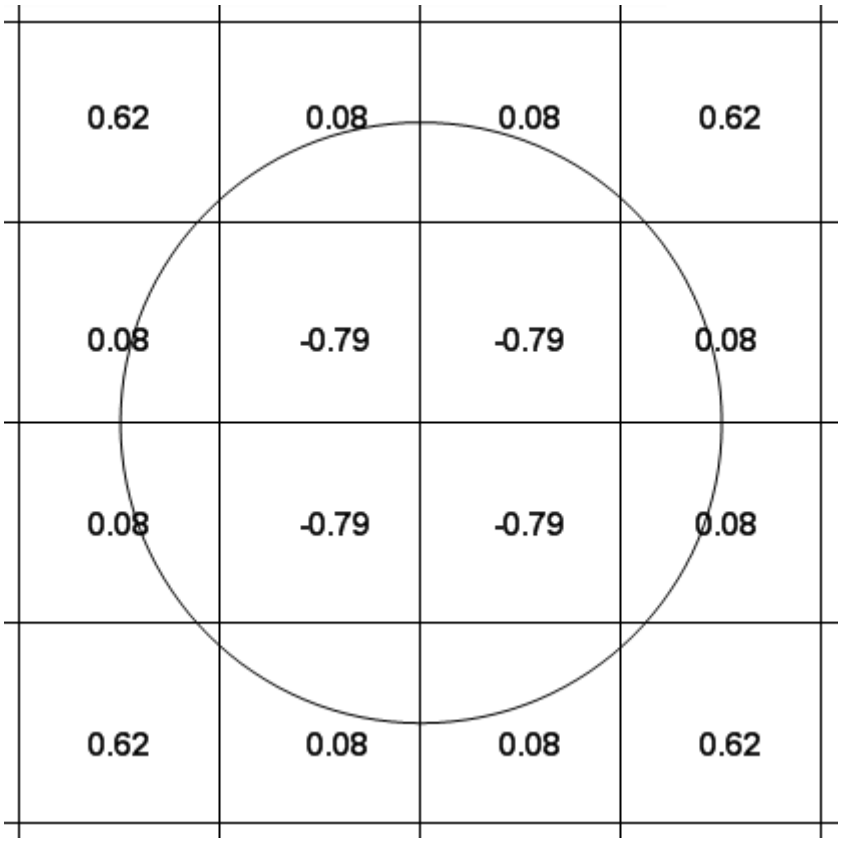


Figure 3.4: The same grid as Figure 3.3 but this time showing the surface contour as well. Even though a circular contour is shown, in actuality, the low resolution of the grid will not be able to the support the circle's continuous curvature. The idea is analogous to the staircase effect in bitmaps.

Such an approach can quickly become very expensive computationally, even on modern processors with multiple cores. Various techniques for efficient storage are described in Chapter 4 and Appendix B.

3.2.2 Visualization via Polygonization

The marching cubes algorithm [53] is able to generate a decent approximation of the original surface with a relatively low resolution grid. The algorithm determines how the surface intersects with a voxel and then uses that information to match it with a look-up table of surface-edge intersections. The table contains 14 patterns for the edges of the voxel intersected for each unique case (Figure 3.5). The algorithm then interpolates the surface intersection along the edge of the voxel and returns vertex position and indices.

The marching cubes algorithm, although a vast improvement from a binary distance field, suffers from the staircase effect (Figure 3.6(a)). The marching cubes algorithm is used with a smoothing pass to align the vertices of the resulting mesh closer to the original surface. This reduces the staircase effect to a more tolerable level (Figure 3.6(b)); although this does increase the time it takes to *march* or to generate the mesh, depending on whether the second pass is applied or not. To obtain smoother surfaces, more passes are required.

Other, similar techniques, have emerged mainly to circumvent the patent that the algorithm held and also to resolve some ambiguous cases where the algorithm would fail to produce proper triangles. Adaptive Tetrahedrizations [55] is one such algorithm which uses tetrahedrons instead of cubes. The algorithm not only eliminates the ambiguous cases, but produces cleaner meshes; although the resulting mesh has a higher density when compared to the marching cubes algorithm. Resolving ambiguities in the algorithm has been the focus of many research papers [56, 57].

A more efficient implementation of the Marching Cubes algorithm, proposed by Lewiner et al. [58], ensures topologically correct results (i.e. eliminates cracks and topological inconsistencies because of the ambiguities in the original algorithm) for any input data.

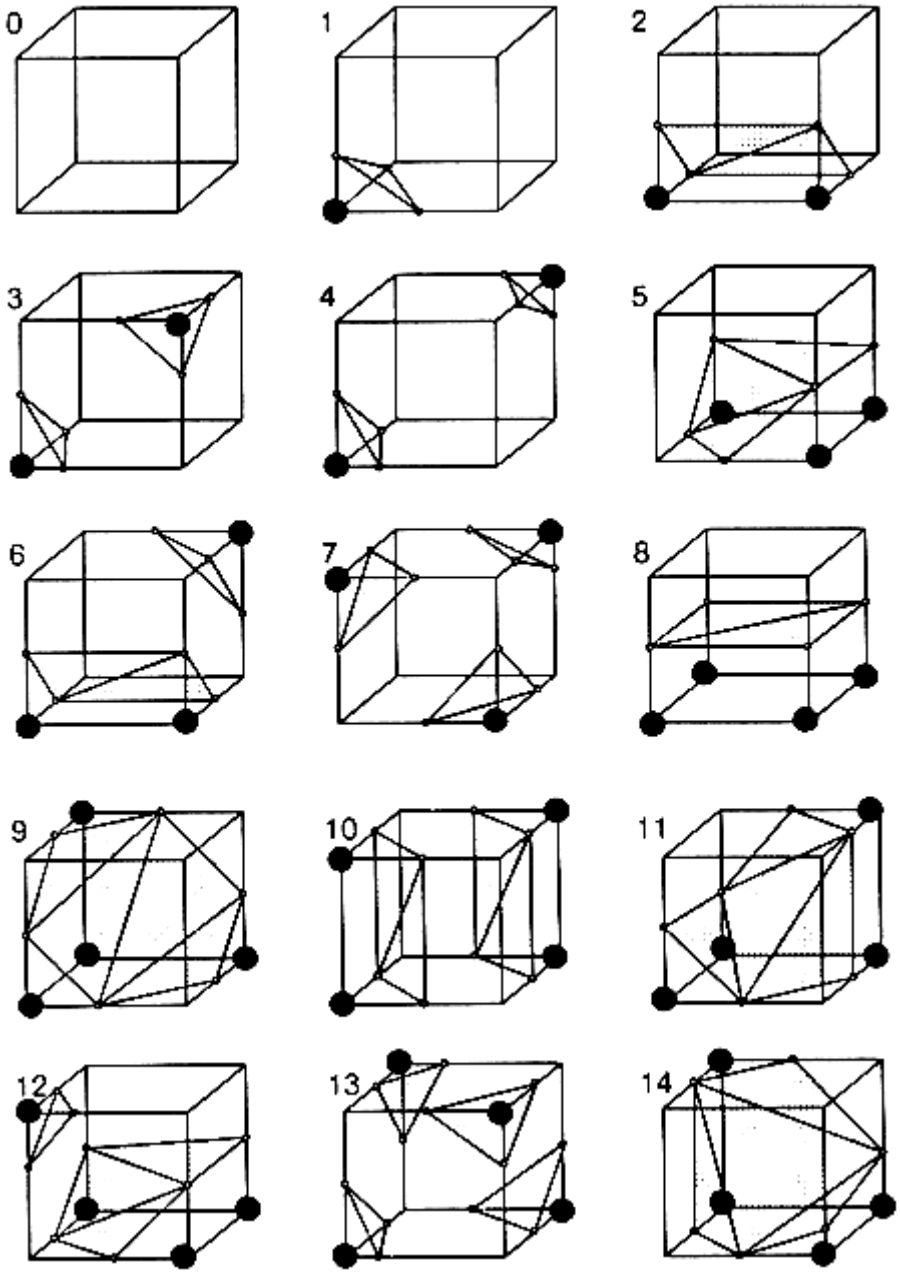


Figure 3.5: The Marching Cubes 15 cube configurations [54]

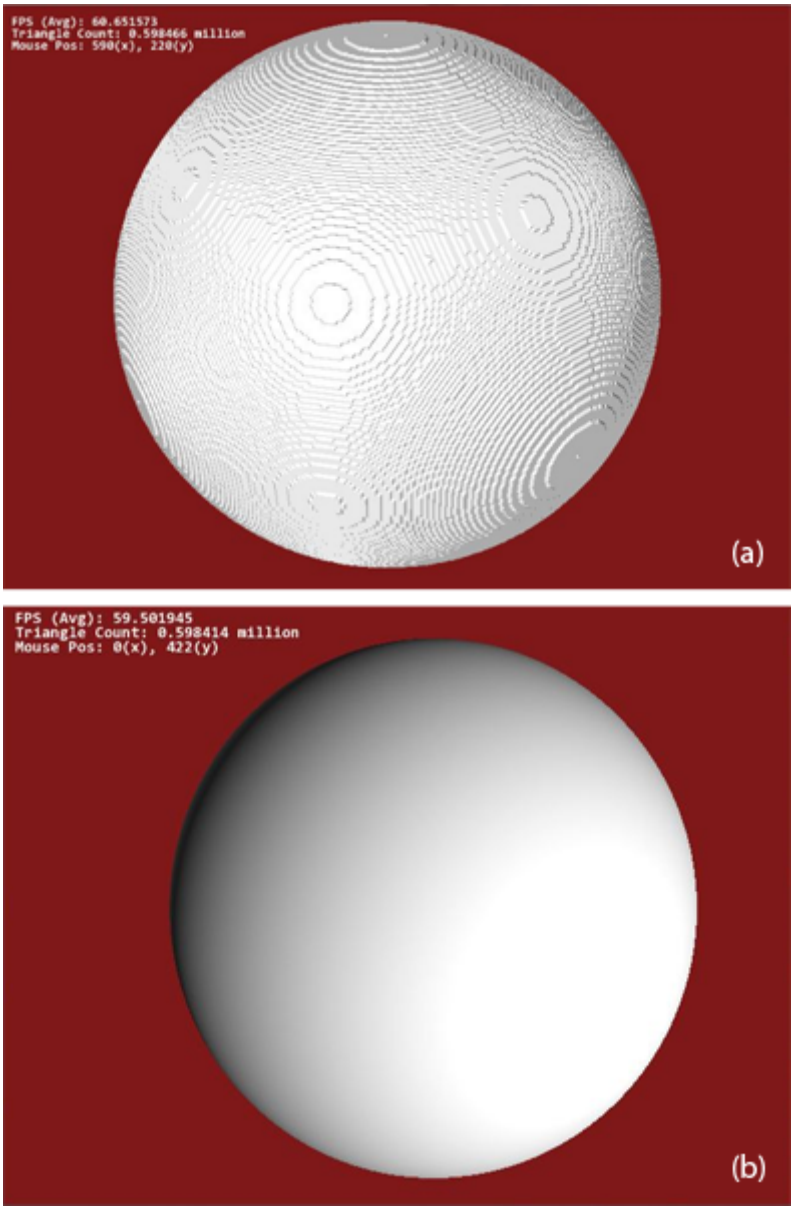


Figure 3.6: (a) a staircase effect produced by a binary distance field. The polygonization is done using Marching Cubes but without extra distance information, the algorithm cannot generate a proper mesh, (b) smoother normals obtained with a smoothing pass that align the vertices of the resulting mesh closer to the original surface. In the current implementation (Chapter 4), a more efficient method is used that does not require an additional pass

3.2.3 Spatial Data Structures

The use of spatial data structures when constructing distance fields is an important technique to reduce memory consumption. Spatial data structures have been researched extensively although their usage may vary depending on the application. Some of the more notable data structures include Octrees, KD-Trees and Adaptively Sampled Distance Fields (ADFs) [38].

The octree are generally a good choice for sparse data sets because of its simplicity. KD-Tree, a spatial tree similar to octree but without cuboid restrictions, can be more efficient than Octrees with sparse data sets but are more complicated in their implementation. When compared to the linear counterparts, octrees provide faster searches and reduce memory requirements.

ADFs improve over the octree by reducing the number of voxels required to represent the same surface. ADFs accomplish this by taking into account surface curvature and discarding voxels that do not experience a curvature change.

3.3 Manipulation of Implicit Surfaces with LSM

Manipulation of an implicit surface is quite different than that of other surface types. For instance, polygonal meshes can be manipulated by control points located directly on the surface. There is no similar manipulation model found in the level set framework. Another method must be used to manipulate the surface.

Figure 3.7 shows a simple signed distance field grid which approximates a circle of unit radius. Even though the grid is an approximation of the contour, it is continuous across the interface. The surface gradient at any voxel can be calculated by querying the neighboring cells. Using bi-linear interpolation the gradient at any point can be calculated within the range of the grid.

The gradient can be changed by manipulating the distance values, effectively manipulating the underlying implicit surface representation. In Figure 3.8 a change in the original contour is introduced by changing the distance values of two voxels. Note that to make it easier to understand, a low resolution grid is shown. A smooth change in the surface contour illustrated in the figure will require a much higher grid resolution.

In general an implicit surface in 2D is defined as

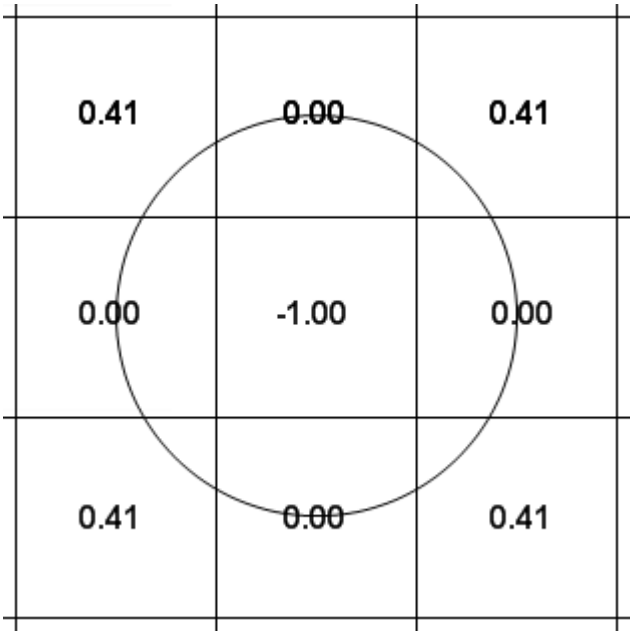


Figure 3.7: A 2D 3×3 signed distance field grid approximating a circle with a radius of 1.5 units where each voxel is a unit length.

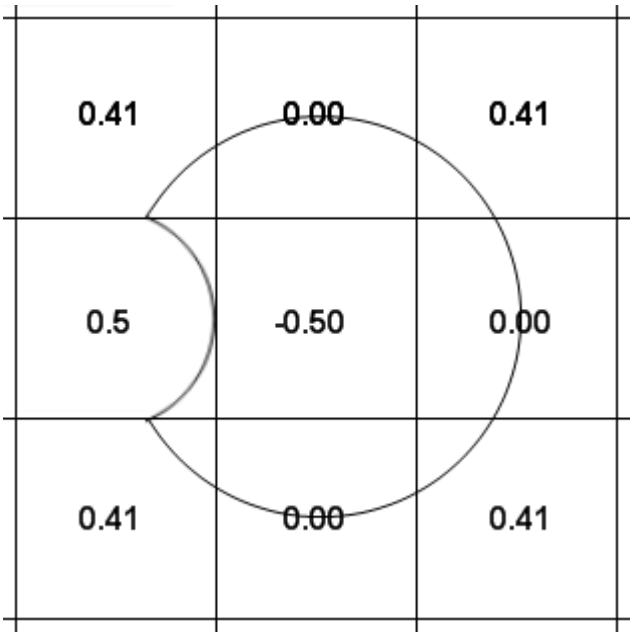


Figure 3.8: Surface contour reflecting the change in the distance values. Note that this is for illustrative purposes only as the distance field shown here has a very low resolution and cannot represent the curve shown.

$$S = \{(x, y) \in R^2 | f(x, y) = 0\} \tag{3.1}$$

and in 3D is defined as

$$S = \{(x, y, z) \in R^3 | f(x, y, z) = 0\} \tag{3.2}$$

where (x, y) is a point in R^2 and $f(x, y)$ is the implicit function. If the surface definition depends on time, the *level set formulation* can be defined as

$$S_t = \{(x, y, t) \in R^3 | f(x, y, t) = 0\} \tag{3.3}$$

where S_t is the surface at time t. The evolution of the surface can be defined by the following integration

$$f_{new}(x, y) \approx f_{old}(x, y) - L(x, y) \cdot \Delta t \tag{3.4}$$

$$L = \vec{V} \cdot \nabla(x, y) \tag{3.5}$$

where $\vec{V}$ is the vector field and $\nabla(x, y)$ the gradient. Solving for L is called *solving the level set equation.*

3.3.1 Advection with Vector Fields

Suppose one wants to move the surface shown in Figure 3.7 to the right without distorting the original contour. This can be done by enveloping the grid with a vector field $\vec{V} = (1, 0)$. The dot product of $\vec{V}$ with $\nabla(x, y)$ will return the change in the interface. The $\nabla(x, y)$ for a vector $\vec{V}$ can be calculated using the upwind scheme [59, 60]

$$\frac{\Delta f}{\Delta x} \approx \begin{cases} f_x^- = f(x+1, y) - f(x, y) \;\; if \;\; V_x < 0 \\ f_x^+ = f(x, y) - f(x-1, y) \;\; if \;\; V_x > 0 \end{cases} \tag{3.6}$$

and

$$\frac{\Delta f}{\Delta y} \approx \begin{cases} f_y^- = f(x, y+1) - f(x, y) \;\; if \; V_y < 0 \\ f_y^- = f(x, y) - f(x, y-1) \;\; if \; V_y > 0 \end{cases} \tag{3.7}$$

The resulting solution of the level set equation can then be applied in Equation 3.4 to get the final distance.

Note that the vector field need not be the same over the entire interface. Variations in the vector field will appear to *sculpt* the surface rather than moving it. The framework is able to perform such sculpting operations by applying a varying vector field across a portion of the surface where the radius of influence is controlled by a *brush size* parameter. For a more detailed explanation and implementation, see Chapter 5.

Calculation of Δt

There is an upper bound on Δt when using Euler integration on the distance field. The only consequence of using too low a Δt is that the desired manipulation will take more steps to complete. Choosing too high a Δt may corrupt the signed distance field such that it no longer remains continuous.

The calculation of Δt for the current step is relatively simple. The maximum allowed Δt is one over the length of the magnitude of the velocity

$$\Delta T < \frac{1}{max\{|\vec{v}|\}} \tag{3.8}$$

Note that in this case $max\{|\vec{v}|\}$ denotes the maximum velocity over the entire vector field. The upper bound is then applied to *all* values in the distance field. Incorrect results are observed if the max Δt is calculated and applied per voxel per step.

Advection in Normal Direction

The *level set equation* (Equation 3.5) performs advection in the direction of the vector. In some applications it is useful to perform advection in the direction of the surface normal. One such use is adding uncertainty to the current map arising directly because of the motion of an autonomous robot. This idea is explained in greater detail in Chapter 5.

Advection in the normal direction is a simplification of a more general advection. Instead of an external vector field, the surface normals are taken and then multiplied by a scalar to obtain the final vector. This calculation can then be performed on the whole interface (or a section of it).

Taking and simplifying Equation 3.5 and substitute the velocity with the surface normal and some speed s

$$L = -\vec{V} \cdot \nabla(x, y) = -s \cdot n \cdot \nabla(x, y) = -s \cdot |\nabla(x, y)| \tag{3.9}$$

where s is the speed, n is the normal of the surface at (x, y) and $|\nabla(x, y)|$ is the magnitude of the gradient.

The gradient can now be calculated as before and use Euler integration to the find the new distance. For a more precise gradient calculation, the *Godunovs method* [61] can be used:

$$(\frac{\Delta f}{\Delta(x, y)})^2 \approx \begin{cases} max(max(f_{x,y}^-, 0)^2, min(f_{x,y}^+, 0)^2) & if \quad s < 0 \\ max(min(f_{x,y}^-, 0)^2, max(f_{x,y}^+, 0)^2) & if \quad s > 0 \end{cases}$$

where f_x^- and f_x^+ are defined in Equations 3.6 and 3.7 respectively.

3.3.2 Normalization

All of the equations presented so far give an approximated result. Any advection performed on the surface corrupts the distance field such that the magnitude of the gradient (which is the length of the signed distance function) is no longer close to one. Consequently, surface advection operations on the distance field will produce

undesirable results. The distance field thus needs to undergo an additional step called *re-normalization*. This step can also repair damage caused by noisy vector fields which is especially important when working with noisy sensors.

The re-normalization is performed by solving the *Eikonal Equation* [62]

$$\frac{\Delta f(x,y)}{\Delta t} = F_s(x,y)(1 - |\nabla f(x,y)|) \tag{3.10}$$

where $F_s(x,y)$, a continuous function, is defined as

$$F_s(x,y) = \frac{f(x,y)}{\sqrt{(f(x,y))^2 + |\nabla f(x,y)|}} \tag{3.11}$$

$F_s(x,y)$ can be substituted in Equation 3.9 to obtain the new *normalized* distances.

There is, however, a problem with the above equation. To calculate $|\Delta f(x,y)|$ (the magnitude of the gradient) $F_s(x,y)$ is required to be calculated in the first place. To solve this issue, the magnitude of the gradient is taken as the distance between two voxels in the signed distance field. Thus, if the signed distance field grid has voxels of unit size, the magnitude of the gradient for Equation 3.11 will therefore be 1

$$F_s(x,y) = \frac{f(x,y)}{\sqrt{(f(x,y))^2 + 1}} \tag{3.12}$$

3.3.3 Surface Normals

The gradient calculated in the past few sections can be used to calculate the surface normal at any point. If the surface is being visualized by polygonizing the SDF, approximate normals for each vertex of the resulting mesh need to be calculated instead of each individual voxel. This is necessary to ensure smoother normals. If the normal of one voxel is applied to all of the faces contained in the said voxel, the resulting polygonal mesh appears to be faceted (see Figure 3.9 for a comparison).

A polygonization method such as marching cubes (see Section 3.2.2) uses interpolation to determine the orientation and placement of the triangles to approximate the underlying implicit surface of a SDF. The technique uses distances from all

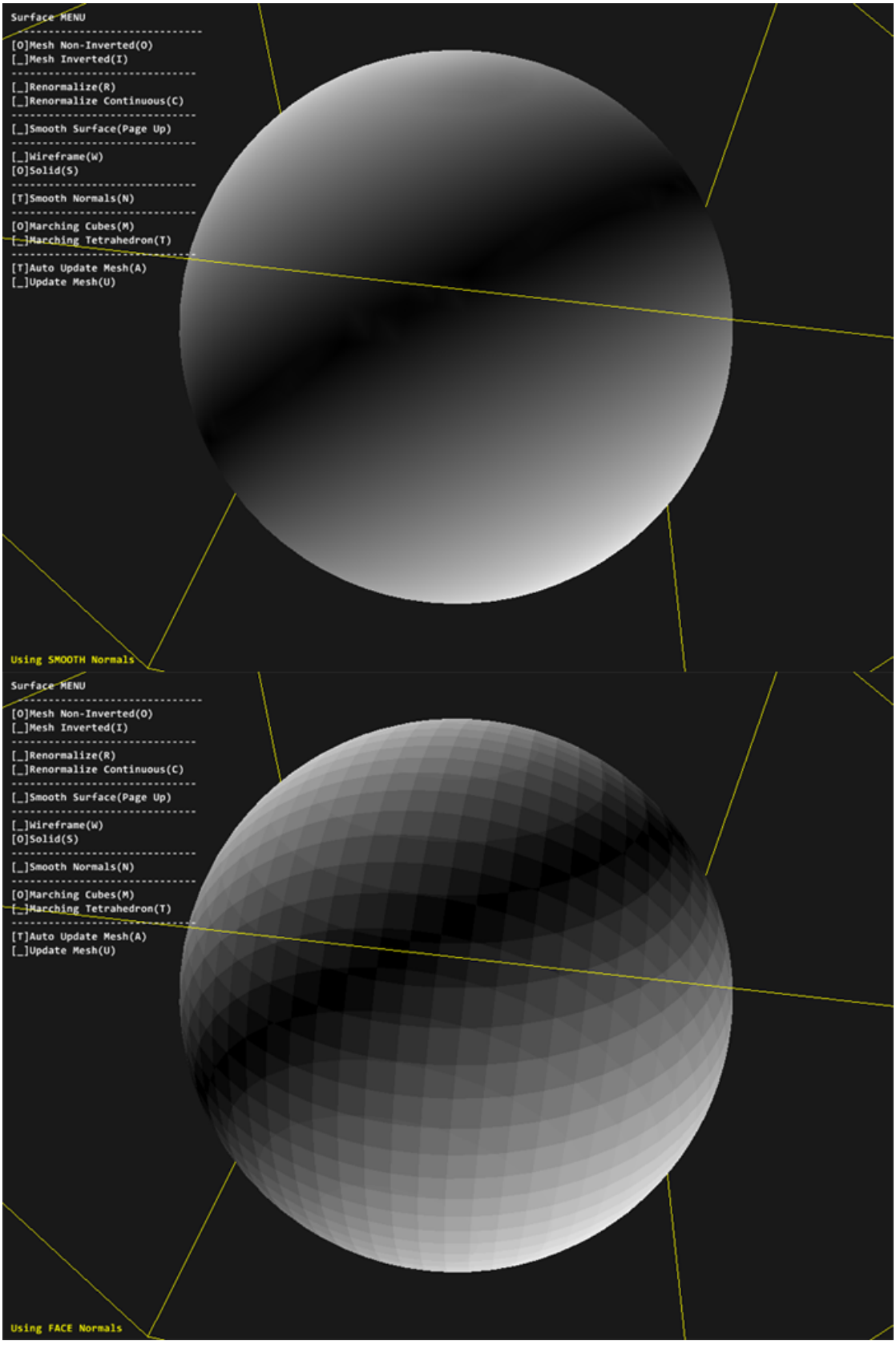


Figure 3.9: A comparison between smooth (top) and hard normals (bottom). The smooth normals implementation calculates the normals per vertex while the hard normals are calculated per face.

corners of the voxel. For the calculation of normals, one can utilize the same distances and tri-linearly interpolate to find the exact normal of each vertex of the triangle calculated by marching cubes or equivalent polygonization method. For details on the implementation, see Appendix E.2.

3.3.4 Smoothing

Many smoothing filters have been developed and are in use in the field of computer vision such as additive smoothing [63], box filter [64], Gaussian blur [65]. Although using such algorithms usually results in some loss of detail, they can also remove some artifacts and make a noisy image or surface appear more visually appealing.

All such algorithms can be applied to help remove artifacts that may arise when manipulating an implicit surface with a noisy vector field. For example, in Figure 3.10 the distance field has developed holes because of insufficient data. Figure 3.11 shows the surface after applying a box blur over the entire grid. The blur operation is able to fill holes because of insufficient data. Better, more robust methods [12] are available for artefact removal and surface reconstruction which can be implemented and applied in a similar fashion.

3.3.5 Narrow Band Method

Implicit surface advection takes place around the area where there is a sign change. Updating the complete distance field does is a direct waste of computational resources because voxels that are not neighbours of intersecting voxels do not affect the surface contour. The voxels that undergo a sign change are updated, called the gamma band, and their neighbors, called the safe tube. Collectively, this method is called the narrow band method, and is described in more detail by Peng et al. [12].

When the interface inside the gamma band starts to change, neighbors from the safe tube are assigned to the gamma band with a distance of +/- half the width of the gamma band (Figure 3.12). Some voxels in the gamma band may enter the safe tube while some safe tube voxels may be discarded altogether if they exit the *safety* of the narrow band.

The safe tube must always be built *after* normalization, otherwise voxels entering the safe tube will force the voxels in the gamma tube to re-adjust.

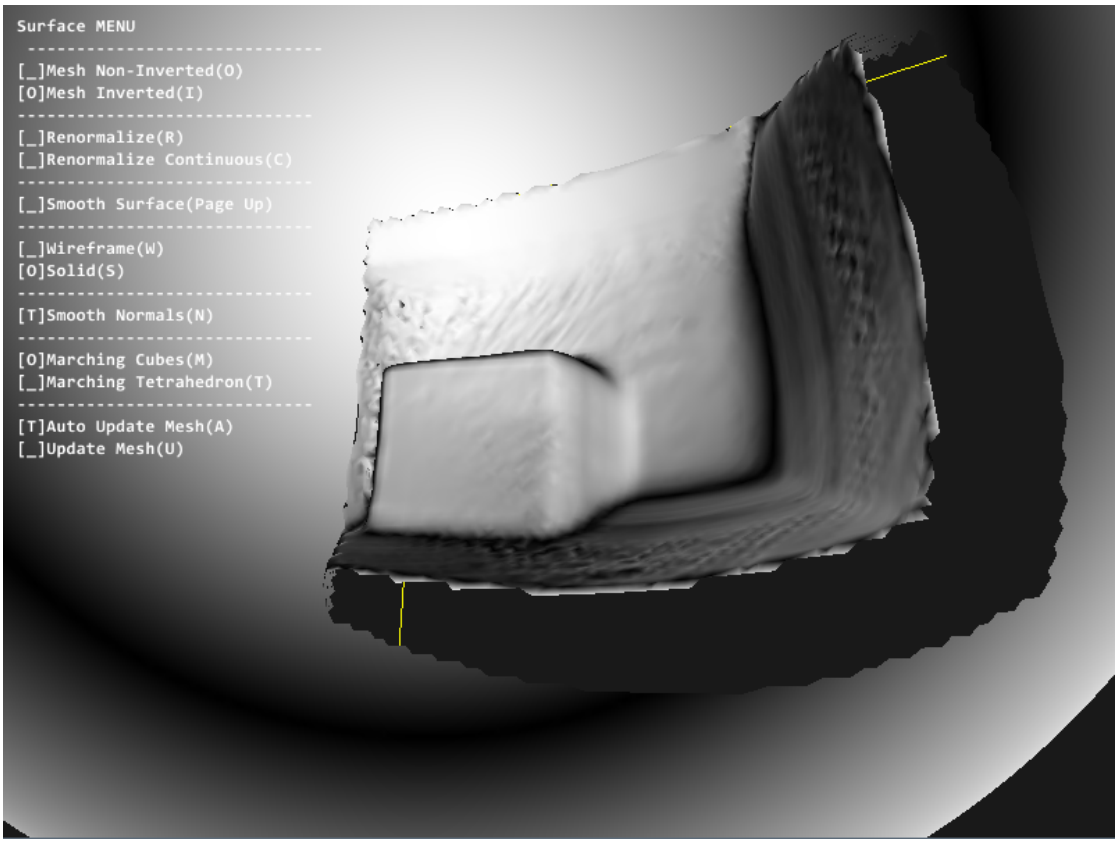


Figure 3.10: Distance field that has developed artefacts and holes where sculpting information was missing.

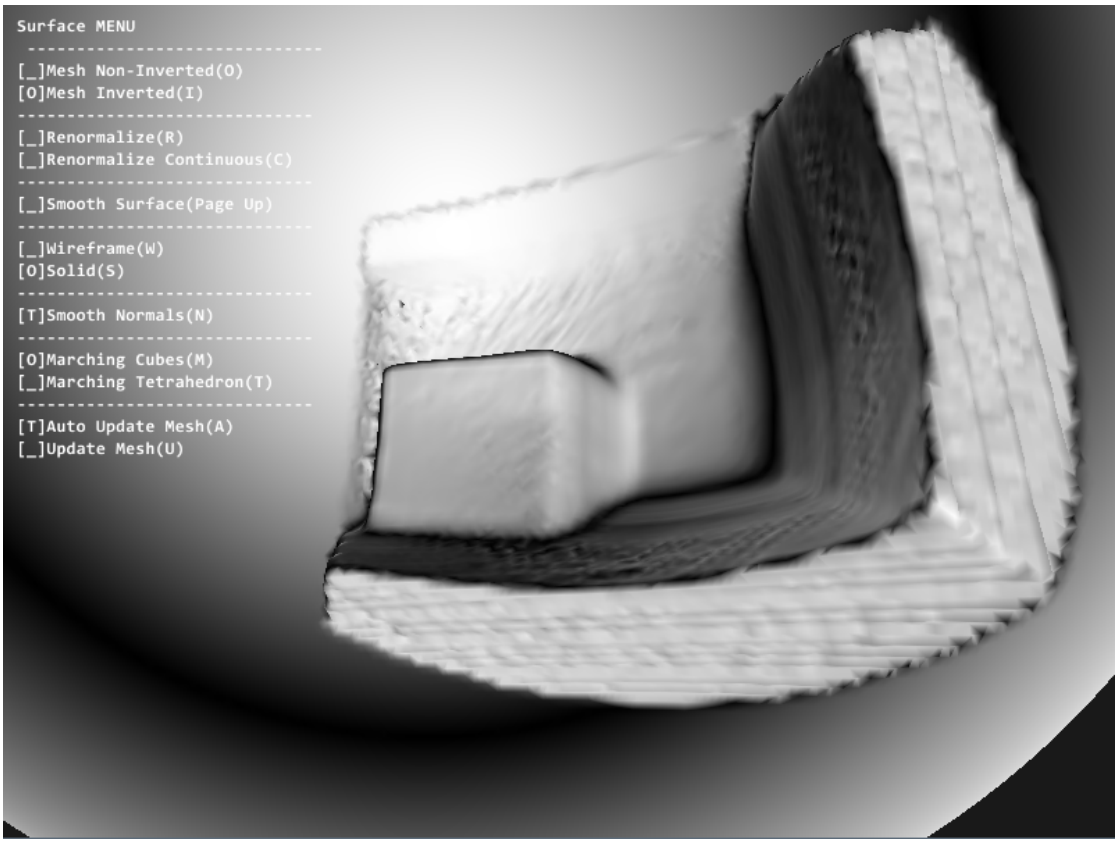


Figure 3.11: The distance field in 3.10 with one blur pass applied. The blur retains 95% of the original surface. Most holes have been removed by this pass.

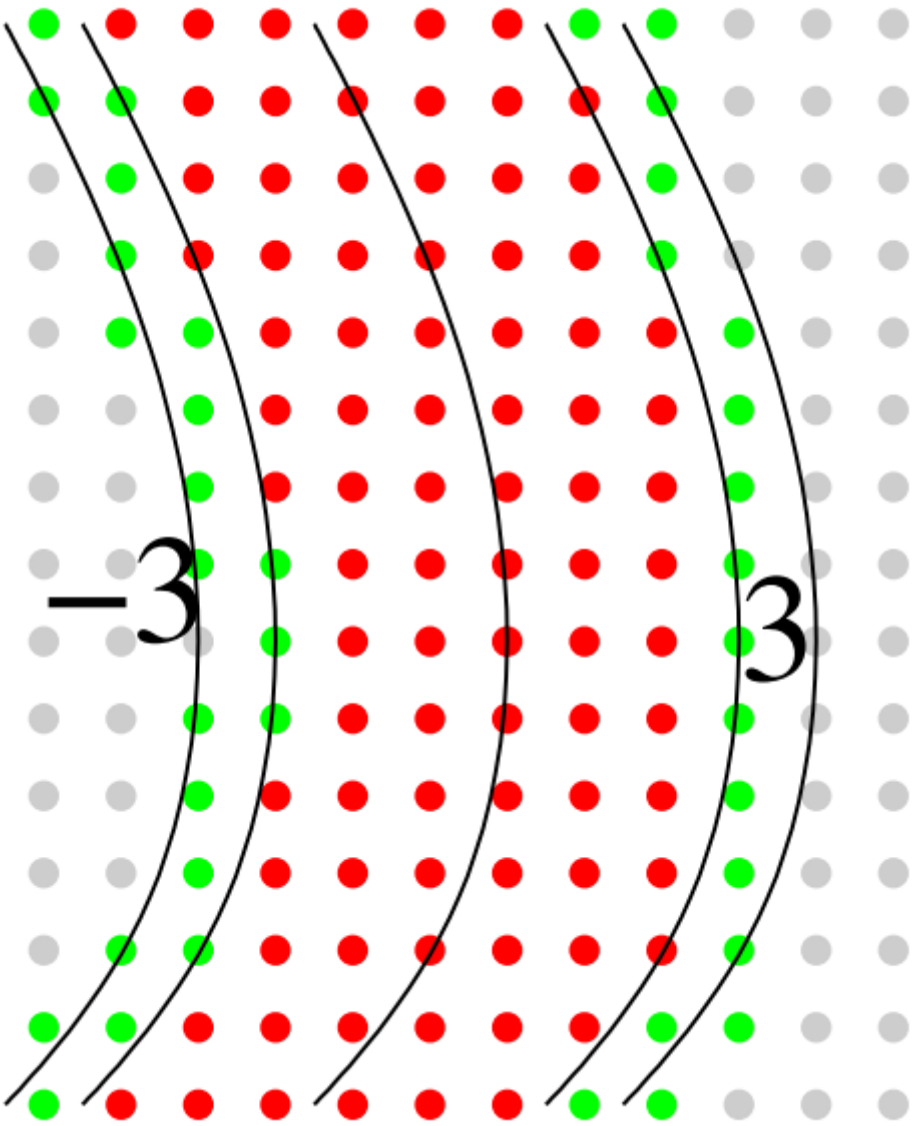


Figure 3.12: The gamma band (red dots) and the safe tube (green dots) making up the narrow band [61].

3.4 Summary and Discussion

Considering the previously mentioned computational requirements, a fair question to ask is what advantages do implicit surfaces, and in turn, signed distance fields offer over parametric surfaces.

Implicit surfaces have several advantages over parametric surfaces which have warranted research in various fields, especially computer graphics. Although a signed distance field is an approximation of an implicit surface, it is still continuous in nature and allows some operations to be performed at a much lower complexity than the parametric counterparts.

Surface intersection of a parametric surface is a complex operation, especially if the surface is complex (e.g. a single piece concave surface). The computational complexity grows exponentially as the complexity of the surface grows. Although there are methods proposed to allow faster intersection of the complex surfaces by decomposing the surface into simpler pieces [66], the complexity remains high at $O(n^m)$. This is due to the nature of parametric surfaces where points can be generated directly on the surface, but never inside or outside the surface. This limitation extends to navigation and path planning, which becomes increasingly more complex as the complexity of the environment itself grows. Intersection tests on an implicit surface can be performed in constant time simply by querying the implicit function (or a *signed distance field*) and examining the returned signed distance. The time complexity remains constant regardless of the complexity of the surface itself.

Constructive Solid Geometry (CSG) is a technique used to obtain a solid non-intersecting surface by addition, subtraction and union of two or more surfaces. Distance fields can perform these operations very easily, no matter the complexity of the intersecting surfaces, while producing smooth results. In contrast, performing these operations on simple parametric surfaces is a challenge and given a complex surface, almost impossible. Another use for fast intersections is collision detection of complex objects, including deforming surfaces [50].

Hole-filling, or surface reconstruction, is another very useful application of distance fields which is again very difficult or impossible to perform with parametric surfaces [12, 45]. Smooth surfaces can be generated from segmented data while preserving fine detail [67]. Errors in geometry that can result from 3D scanners, even high resolution laser scanners, can be filled [68].

Deformation of surfaces, including a complex operation called morphing, can be done very accurately and easily with distance fields [69]. Morphing between two parametric surfaces is a complex operation and does not always result in a clean surface, especially if the vertices of the morphing surfaces do not match. As will be shown in a later section, morphing between two implicit surfaces via distance fields produces accurate, believable and predictable results.

Another major advantage not shared by parametric surfaces is level of detail. An implicit surface, combined with a data structure such as an octree, can have an unlimited level of detail provided memory requirements are not an issue. The spatial tree can be subdivided to many levels where each level can be manipulated as a separate tree thus keeping the computational requirements very similar to that of the original, un-subdivided tree. Variational detail offered by spatial data structures and distance fields may allow faster processing of information of mapped environments in SLAM algorithms.

The above mentioned advantages warrant extensive use of implicit surfaces and distance fields in various applications, especially computer graphics. Unfortunately, distance field generation is computationally very expensive and storing a high resolution distance field requires too much memory to warrant use in applications such as real time graphics.

As discussed previously, computers cannot represent continuous surfaces. The surface must be discretized which results in a loss of resolution and quality. For a good approximation of the implicit surface, a high resolution grid must be used which will inevitably utilize a lot of memory and will take too much time to iterate. Various methods have been proposed to alleviate this particular problem with specialized data-structures [70, 71, 72].

In this thesis, the disadvantages outlined here will be addressed and a different approach to mapping environments in SLAM is proposed that has not been explored previously. A few different frameworks will be presented that have been developed to show successful mapping of environments using the proposed approach in the following chapters.

Chapter 4

Proposed Methodology

Rather than representing the world explicitly with salient features, the world is represented as an implicit function. With this representation, SLAM can be redefined in a coherent manner providing the opportunity to handle any differentiable and dynamic process.

The implicit function is defined as a signed distance field represented as a grid over the area that is to be mapped (see Figure 3.1). Within each cell the scalar distance to the closest surface point is stored. This distance is signed where the sign represents the locality of the cell relative to the surface. Operations on the distance fields can be defined using the level set framework [73]. Distance fields and level sets have a long history in the computer graphics and computer vision literature representing 2D and 3D surfaces [34], tracking deformable objects [42] and producing highly realistic animation of complex propagating surfaces such as fluids, fire or smoke [74].

The following sections summarize the salient points of Chapter 3 and provides details of the approach to re-defining the SLAM mapping problem.

4.1 Level Set Methods

As discussed in Chapter 3, level set methods provide a mathematical toolbox that can represent complex dynamic surfaces and deformations on these surfaces. In this section the level set framework is presented and shown to deform an implicit surface.

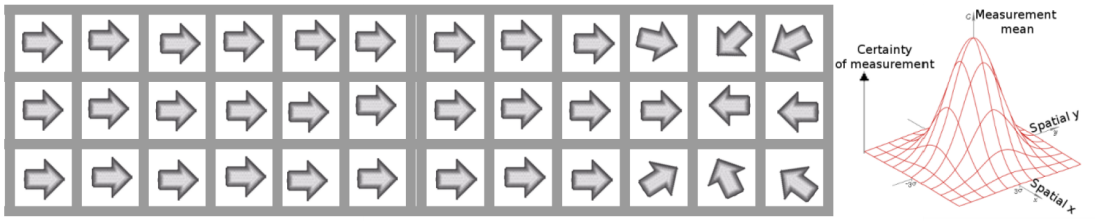


Figure 4.1: Measurement Velocity Fields. (a) A uniform velocity field that would push the surface to the right (b) A velocity field that pushes the surface towards the right but then converges to a particular area since the velocity directions change (c) A Gaussian model that could be used to represent certainty of the measurement model. Note that a gaussian function is a scalar field and the gradient of the Gaussian will produce a velocity field that points towards the maximum (or mean).

Surface evolution is achieved by applying an external velocity field to the distance field. As seen in Chapter 3, the surface is defined as:

$$S(t) = \{(x, y) \in R^2 | \theta(x, y, t) = 0\} \tag{4.1}$$

where (x, y) is a 2D point on a Cartesian grid and $\theta(x, y, t)$ is the distance field function that is deforming over time, t. The surface is defined on the locations where $\theta(x, y, t)$ is equal to zero.

Evolution (or deformation) of the surface can be defined by solving the following relation

$$\theta_t + \vec{V} \cdot \nabla\theta = 0 \tag{4.2}$$

where $\vec{V}$ denotes an external velocity field and $\nabla\theta$ denotes the spatial gradient of the distance field. The solution to this equation can be derived using a forward Euler integration approach (or a more stable 4^{th} order Runge-Kutta approach) as

$$\theta_{t+\Delta t} = \theta_t - (\vec{V} \cdot \nabla\theta_t)\Delta t \tag{4.3}$$

using an upwinding scheme to compute the appropriate sign of the spatial gradient as described in [73]. Note that in this framework, the only way to deform the

surface is to define a velocity field over the spatial area. Thus, all operations in the variational SLAM framework must be defined in this way.

4.2 Redefining Sensing as an Act of Deformation

When redefining SLAM in the implicit mapping framework, the sensing process must be redefined as an act of surface deformation. For the context of this thesis, the focus will be on using range measurements from laser scanners and stereo cameras but the algorithm developed is easily modified to accommodate general sensors as long as they constrain the map spatially. Assuming a prior estimate of the distance field is available, each measurement constrains a local minimum of the distance function. Figure 4.2 shows a simplified version of the proposed approach.

Since the measurements are assumed to be part of a surface that is to be represented, the distance to the surface at this measurement must be zero (or near zero because of the uncertainty of the sensing process). If the corresponding grid cell in the distance field is not zero, then the function representing the surface must be modified to accommodate this constraint. More importantly, all of the cells between the sensor location and the measurement must be free from obstacles because of line-of-sight restrictions imposed by the sensing process.

To re-define sensing in the level set framework, the velocity field is defined as a measurement that extends over the affected spatial region (see Figure 4.1). The velocity of the measurement is defined (or can be sampled) at each point in the level set grid to determine how the current surface must be deformed to conform to the new measurements (see Figure 4.3 which shows the 2D prototype of the framework. Implementation and results of the 3D framework are provided in Chapters 5 and 6).

By defining a scalar distance field for the measurement the surface can be easily deformed. Note that the gradient (spatial derivative) of a distance field defines the velocity towards the surface which can be used as an external velocity field during the level set formulation. Defining the sensing process in this manner allows encoding uncertainty in the sensor. This approach has much similarity to [75] in that the sensor is essentially a tool that operates on the level set function.

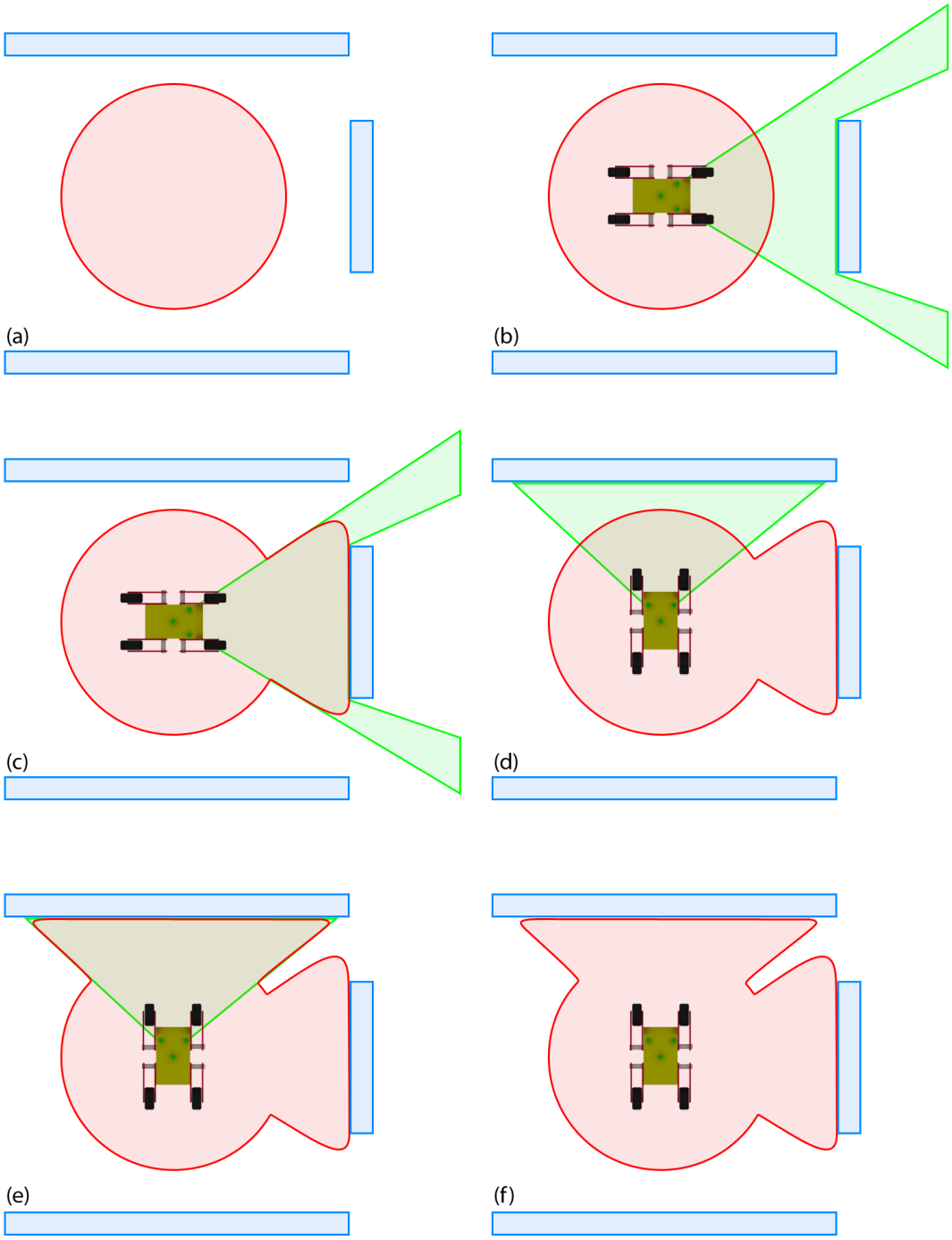
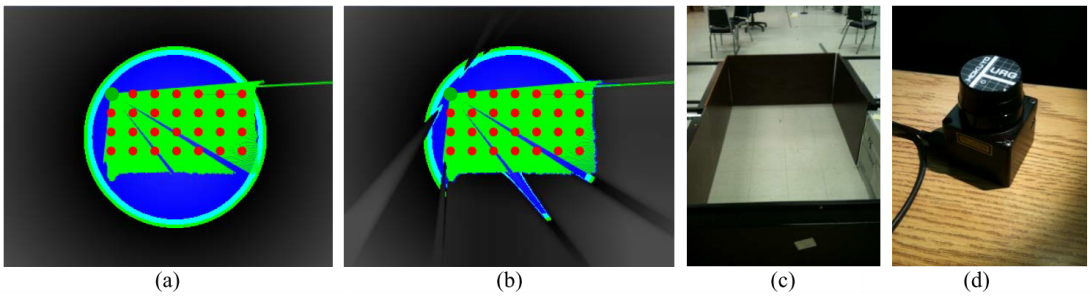


Figure 4.2: (a) An implicit surface (circle) placed in the middle of the environment denoted by blue rectangles (b) The robot is placed at the center of the implicit surface and senses the environment via a range sensor (c) The implicit surface is deformed to conform with the measurements taken from the range sensor (d) The robot turns North and senses the environment again via the range sensor(e) The implicit surface is deformed again, (f) the final deformation with respect to the ranged sensor readings.



(a) (b) (c) (d)

Figure 4.3: Sensing as Deformation of a Level Set. The green lines represent the laser measurements, the blue area is the interior of the contour (where the distance field is positive), the light blue areas show the contour of the surface (where the distance field is close to zero). (a) Initial distance field showing laser measurements (b) after deforming the distance field (c) Shows the experimental setup in the lab for the vehicle motion (d) The laser scanner used in all the data shown in this paper. In (a,b) grey regions are outside of the surface (positive distance to the surface), blue represents the interior of the surface (negative distance) green with aqua colored regions represent the zero level set and the green lines represent the laser line scan data. The red dots represent the locations in which the sensor was placed in the world and the green dot represents the current location of the sensor for this set of measurements.

4.3 Implicit Surface Modeling

Just as the level set method enables the representation of complex surfaces that are difficult to define analytically, the implicit sensor modelling approach can define arbitrarily complex sensor models which can be estimated through traditional calibration processes. Thrun [1] defines sensing as a complex interaction of multiple probabilistic models. In a similar fashion, the implicit sensor model can accommodate noise, complex geometry and spatially varying parameters.

4.4 SLAM and the Level Set Framework

In probabilistic SLAM, the posterior to estimate is defined as:

$$p(s_t, \Theta | z^t, u^t, n^t) = p(z_t | s_t, \Theta, n_t) \int p(s_t | s_{t-1}, u_t) p(s_{t-1}, \Theta | \{z, u, n\}^{t-1}) ds_{t-1} \quad (4.4)$$

where s_t is the robot location at time t, Θ is the map, z^t the measurements, u^t the control inputs and n^t the data associations.

Standard EKF based solutions model the partial differential equations (PDEs) above as a Gaussian with zero-mean noise (see Chapter 2 for more details). The estimation process occurs using a prediction-correction feedback loop where the prediction stage applies the robot motion model and control inputs to determine the predicted map, pose and uncertainty. The correction stage uses the given measurements and data associations to update the predicted map and robot pose by weighting the error between predicted and actual measurement appropriately with current estimates of the uncertainties in the system and sensor.

4.4.1 Sensors

In this approach, each measurement can be defined as an external velocity field to be used in the level set formulation to deform the estimated surface. Thus, there is a direct relation between the probabilistic sensor model, $p(z_t|s_t, \Theta, n_t)$, and the implicit sensor field, S_v. In SLAM, the sensor model is used to define the uncertainty of the measurement during the correction phase of the standard EKF model. If the uncertainty of the measurement is high, the algorithm favors the existing corresponding map location while if the uncertainty is low (relative to the map location), the algorithm weights the measurement higher updating the map feature accordingly.

In this approach, the surface is updated through an act of deformation that depends on the velocity field applied to the surface. If the sensor has a high uncertainty, the velocity will be lower and will have little impact. If the sensor has a low uncertainty, the velocity will have much greater impact on the deformation process. In this framework, the uncertainty of the measurement can be specified over the entire space enabling the possibility of using a complex nonlinear uncertainty model.

4.4.2 Robot Motion

If the robot motion model is redefined, $p(s_t|s_{t-1}, u_t)$, as a tool that deforms the surface in a similar fashion to re-defining the sensor model, the robot motion can easily incorporated into the level set framework. This is accomplished by defining a velocity field over the entire map to incorporate the uncertainty achieved when moving (see Figure 4.3). This robot motion field, R_v, is defined over the entire grid area and is computed by estimating how much each grid cell should move given the robot motion model. Uncertainty is incorporated in a similar fashion as in the

sensor field by weighting the velocities at grid locations by amounts proportional to the uncertainty model.

4.5 Level Set SLAM

Putting together both the discussed robot motion velocity fields, R_v, and sensor velocity fields, S_v, a parallel to traditional EKF SLAM and a mapping algorithm can be defined. The algorithm follows the traditional Predict-Correct feedback loop as follows:

4.5.1 Step 1: Prediction

The goal of the prediction stage in traditional SLAM is to define a prediction of where the robot will be given the motion model and control input which modifies the uncertainties of the map locaitons and robot pose accordingly. The approach taken is an act of deforming the map to incorporate the robot motion field. This is accomplished by solving the level set equation until convergence using the robot motion velocity field. The resulting map will be a representation of the world relative to the robot. The level set equation can be solved in terms of the robot velocity field:

$$\Theta_{t+1}^{-} = \hat{\Theta}_t - R_v \cdot \nabla \hat{\Theta}_t \tag{4.5}$$

4.5.2 Step 2: Correction

The next step is to correct the predicted state with the given measurements. The predicted map can be defined according to the sensor field as discussed above until convergence. This will deform the surface until the zero level set minimizes all of the geometric constraints imposed by the sensor field. The result of this stage is a new map whose zero level set has been pushed/pulled towards the area of minimal uncertainty defined by the sensor field. Similarly, this is defined as a solution to the level set equation using the sensor velocity field as:

$$\hat{\Theta}_{t+1}^{-} = \Theta_{t+1}^{-} - S_v \cdot \nabla \Theta_{t+1}^{-} \tag{4.6}$$

Note that the sensor field can incorporate any geometric constraints imposed on the map. For instance, one can incorporate smoothness constraints based on neighboring values in the distance field, or constraints based on curvature. More importantly, S_v can be used to encode the uncertainties in the sensor.

Chapter 5

Implementation

5.1 Introduction

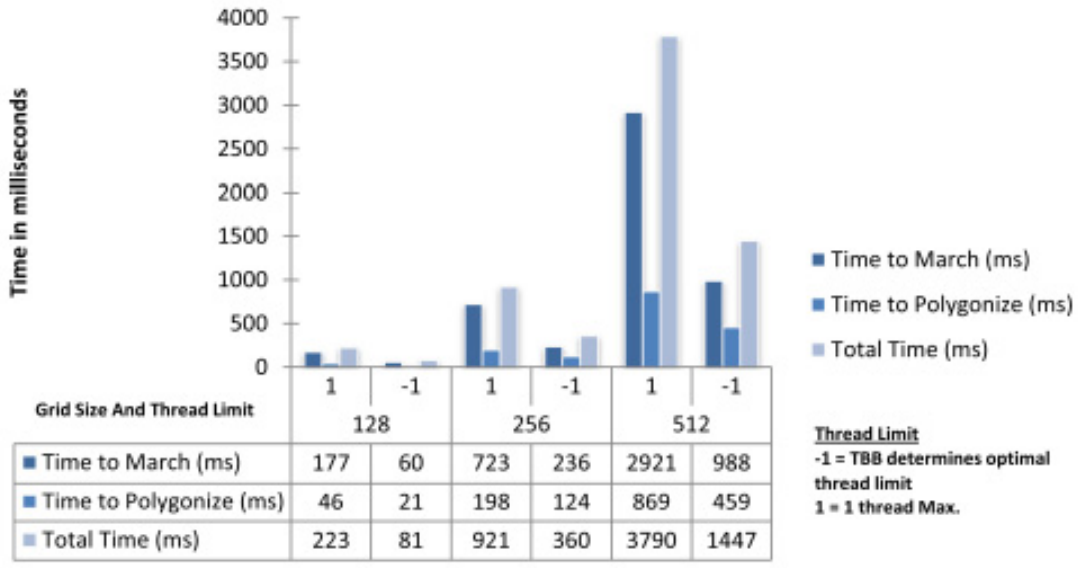
Three successive implementations have been developed to tackle the challenges with LSM highlighted in Chapter 3.

The first implementation developed demonstrates the use of the Octree data structure to reduce memory consumption and to allow for scalable parallel computing on distance fields. In addition, the use of a multi-threaded, highly scalable memory manager is for voxels is demonstrated in the spatial tree. The memory manager virtually eliminates memory fragmentation and guarantees stability of the program when computing operations on the distance fields for long periods of time.

The second implementation demonstrates, in 2D, the predict-correct SLAM algorithm and deformation of a surface contour by a laser sensor. The prototype successfully applies the level set methods on a distance field where the sensor acts as a deformer.

The final implementation extends the 2D framework to allow surface deformation in 3D. The 3D framework is also able to deform the original surface via a set of unordered points (e.g. data from the Microsoft Kinect sensor). Although in this thesis, the deformations in the 3D framework are shown to work with a linear grid, it has been extended to take advantage of the octree spatial data structure.

The level set framework depends on a custom framework and 3^{rd} party dependencies described in more detail in Appendix A.1.



Grid Size And Thread Limit	128		256		512	
	1	-1	1	-1	1	-1
Time to March (ms)	177	60	723	236	2921	988
Time to Polygonize (ms)	46	21	198	124	869	459
Total Time (ms)	223	81	921	360	3790	1447

Figure 5.1: Comparison between single and multi-core algorithms for generating implicit surfaces using marching cubes (lower is better)

5.2 Distance Field Generation and Morphing

The goal of the first level set framework was to allow real-time morphing between two distance fields. The framework is capable of generating a distance field from a function, populating an octree with the said distance field, polygonizing the distance field using the Marching Cubes algorithm and finally drawing the mesh on screen in approximately 81 milliseconds.

The developed framework uses Intel Threading Building Blocks (Intel TBB) to allow the application to use multiple cores. Using Intel TBB along with a custom memory manager and algorithms designed to take advantage of parallel processing, large grids are generated and manipulated in real-time with very low initial surface generation times (see Figure 5.1).

The results of interactive morphing are promising in terms of performance and memory requirements (see Table 5.1). The tests were performed on a quad-core Intel Q9550 processor[1] with 8GB of DDR2 800 MHz RAM. Although the processor's architecture was 64-bit, the program was compiled with 32-bit binaries putting an upper limit on the amount of memory the application can consume regardless of the total available memory of the system. The voxel size was $40 - 72$ bytes (depending on whether a voxel was subdivided or not) without mesh visualization. With mesh visualization turned on, the size increased by $36 - 144$ bytes

[1]Each core of the processor runs at 2.83GHz and shares 12MB of L2 cache

Grid Size (n^3)	Mesh Generation	Avg FPS	Voxel Count	
			Min.	Max.
0	No	372.1	0	0
32	Yes	59.3	2,953	4,681
32	No	278.1	2,953	4,681
64	Yes	39.4	13,769	18,761
64	No	78.7	13,769	18,761
128	Yes	9.5	54,154	74,121
128	No	19.0	54,154	74,121
256	No	5.1	210,121	296,713

Table 5.1: Performance measurements of implicit surface morphing. The mesh generation column specifies whether a parametric surface was generated from the distance field for visualization. These tests were performed on the Intel Q9550 quad-core processor with DDR2 800 MHz RAM.

depending on the number of triangles calculated by the marching cubes algorithm. This supported the assumption that it is possible to create and manipulate a 3D distance field in real-time (in this thesis, greater than 10 FPS is considered *real-time* for surface manipulation and greater than 3 FPS is considered *interactive*).

The details and breakdown of the data structures including multi-threaded implementations can be found in Appendix B and D.

5.3 2D Level Set Deformation

The 2D level set framework was built purely to demonstrate the approach outlined in the previous sections. In the 2D framework, each pixel was taken as a unique voxel. Surface deformation was done with the methods outlined previously. This demonstrates the prediction-correction SLAM algorithm works well in a 2D LSM framework. The same concepts in this framework were then extended to the 3D real-time level set deformation framework described in the next section.

The results of the tests ran with a laser sensor with the 2D LSM framework are provided in Section 6.1.2.

5.4 Real-time 3D Level Set Deformation

The final step is to extend the 2D algorithm to 3D and demonstrate its use with real-world sensor data. The current framework is able to work with both a linear grid (Figure 5.2) and an octree spatial grid (Figure 5.3). First, a sculpting tool was

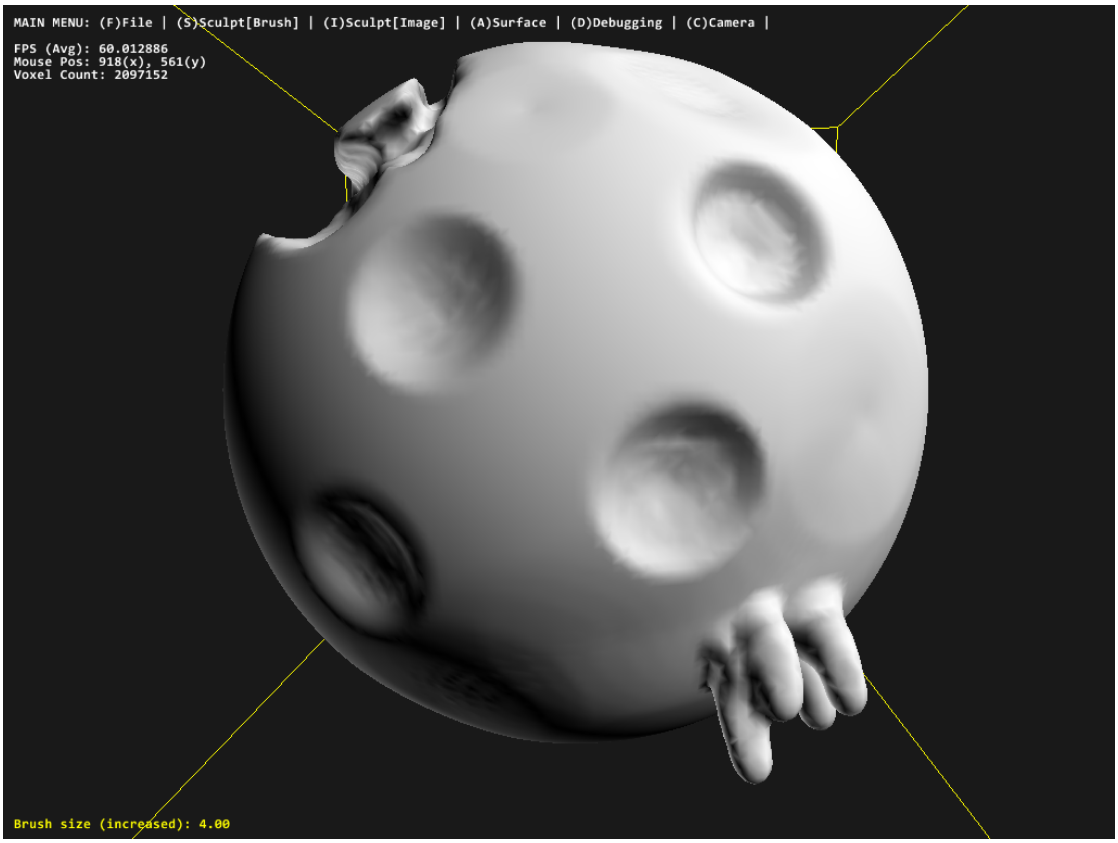


Figure 5.2: A relatively complex sculpt was achieved with ease with the default sculpting brushes (linear grid shown with a resolution of 128^3 with a memory usage of). Note the extrusions on the bottom right of the surface; the surface can not only be deformed but can *create* new areas of the surface without the possibility of the surface tearing.

developed using data structures including a linear grid and an octree grid. The octree grid is able to achieve interactive frame rate for surface sculpting with small brush sizes. Larger brush sizes (5+ units) are limited by slower neighbor lookups in the octree implementation although for larger grids (128^3) with large brush sizes (20+ units) the linear grid implementation can no longer sculpt interactively (a brush size of 1 requires 26 neighboring voxels to participate in the surface advection operation).

Apart from direct sculpting via a brush tool, the framework is capable of deforming the surface to conform to a set of unordered points. Figure 5.4 shows a depth map rendered from Maya[2] which is then projected onto an inverted sphere in the framework. The sculpt takes place interactively over a period of a few frames. The result is shown in Figure 5.6. The deformation was performed on a distance field grid with a resolution of 128^3. The resulting distance field is stable and free of any holes.

[2]3D modeling software from Autodesk

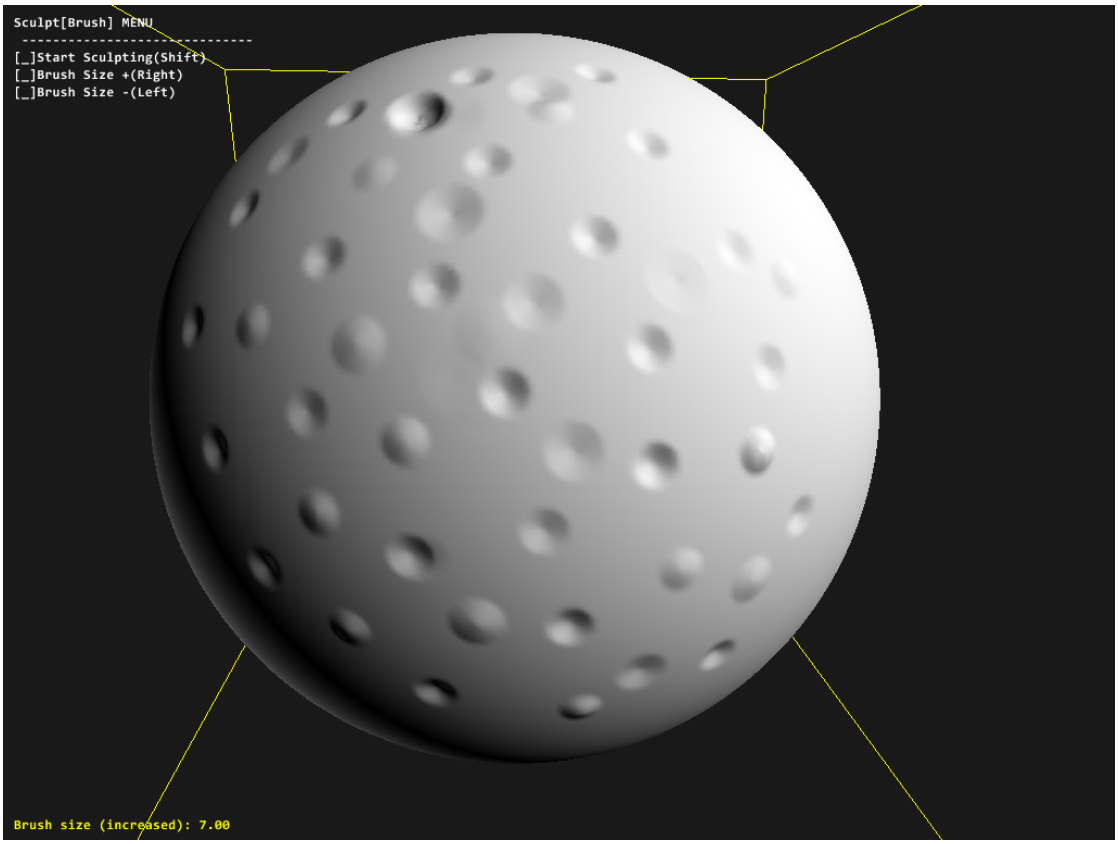


Figure 5.3: The same sculpting brushes used here in the octree implementation. The grid resolution is 256^3 with memory usage of 541 megabytes. Interactive sculpting is achieved with a brush size of seven units and real-time sculpting with a brush size of five and below.

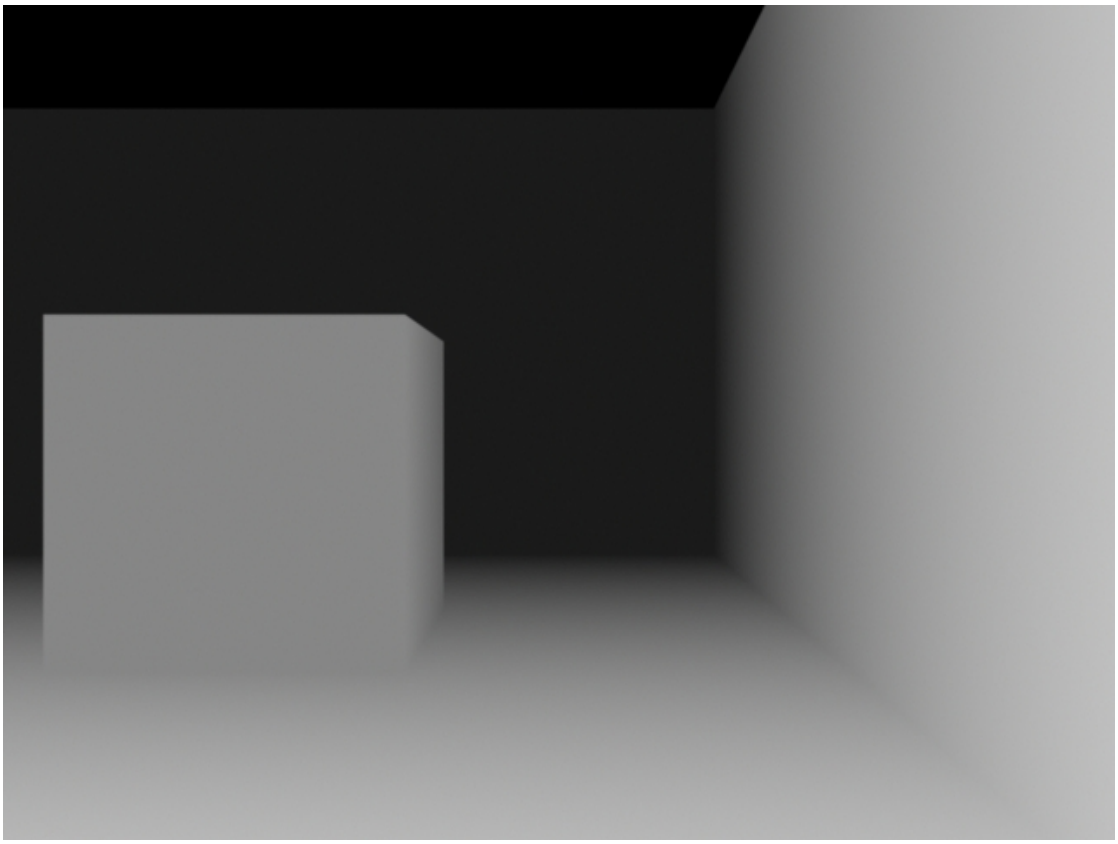


Figure 5.4: A depth render from the 3D scene in Autodesk Maya. With the projection and transformation matrix of the camera, points can be projected into a 3D scene. Note that this particular depth render is incorrect as the near and far planes are set automatically (and incorrectly) by the built-in shader (even though the camera's near and far planes are set properly) which caused some confusion during deprojection operations. See Figure 5.10(b) for a proper depth render.

Figure 5.5: Left: A screenshot from the scene in Maya, Right: The depth render of the scene. This render is used to deform an implicit surface (Figure 5.6).

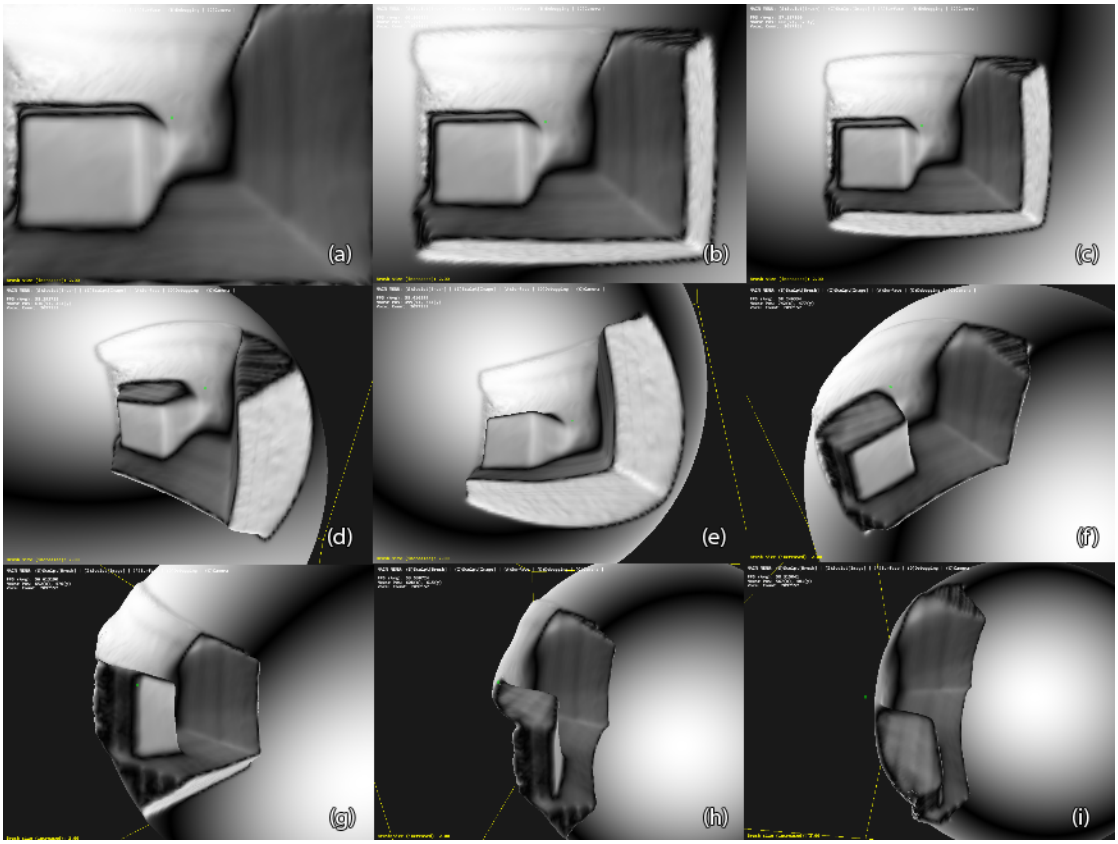


Figure 5.6: The results of surface deformation of the depth map from Figure 5.5, (a) Surface advection using the projection of the depth map shown in Figure 5.4. The initial surface was an inverted sphere (b)-(i) The camera is moved to show the deformation from different angles The surface outside of the frustum in (a) is stable and free of any holes.

The developed framework uses C++ templates to allow some decisions to be made at compile time. This includes the selection of grid types (linear vs spatial). A policy based architecture is used which allows extending the framework with other grid types (such as kd-trees), again, incurring no run-time overhead.

5.4.1 Voxels

```
// T_Creator is the grid that creates Voxel. It is required for
// discovering the types. T_Parameters is the optional user
// defined parameters that the Voxel class can store in addition
// to the signed distance
template <class T_Creator, class T_Parameters = VoxelParamsEmpty>
  class Voxel
  {
  ...
  }
```

The voxel object is a simple class that stores the signed distance as well as user defined parameters. In the current implementation the voxel size is kept small but at the same time some optimization tricks were not used in an effort to keep the implementation from becoming too complicated. The voxel, through the voxel parameters, stores:

- The voxel's grid index[3]
- Temporary storage for new signed distance values. This is useful when performing sculpting operations, since old distance values of neighbors are required, where the neighbors may have their new distance values calculated in a previous iteration)
- A *bool* variable to signify if a voxel intersects the surface
- A *std* :: *vector* <> to store triangles returned from a marching algorithm[4]

The voxel's total size is 32 bytes (the *bool* variable is padded by the compiler to occupy 4 bytes).

[3]The grid index is stored with the help of three *ints*. This can be reduce that to one *int* and the number converted to a 3D coordinate, which would save an additional 16MB of memory.

[4]a *std* :: *vector* <> stores 3 pointers and therefore has size of 12 bytes

Only one signed distance per voxel instead of eight is stored, one for each corner of the voxel. To get the signed distance of each of the corner voxels, neighbor look-ups are performed on the grid to get the desired signed distance.

5.4.2 Linear Grid

The linear grid implementation stores voxels in a *std* :: *vector* container. It also has some (shared) functionality which allows it to return voxels intersecting a ray and a frustum.

The linear grid is used for all the experiments in this thesis. The grid implementation makes no attempts at saving memory (e.g. run length encoding (RLE) compression) and therefore the resulting grid is not *sparse*. Fast neighbor lookups and cache coherent iteration is achieved at the cost slower search operations.

5.4.3 Spatial Grid (Octree)

The octree spatial grid implementation has branch and leaf nodes (see Appendix B for details on how an octree works). There is a special branch node, the root node, which can be accessed by the grid at all times. The voxels are stored in a leaf node whereas the branch nodes store child nodes (if they exist).

Each of the eight main branches (children of root) store a set of intersecting voxels in linear containers as well a pointer to a mesh object. This is to ensure fast iteration when applying a marching algorithm. Without storing the intersecting voxels, the marching algorithms will be required to search for the leaf nodes in sequence which is a costly operation, especially in a critical loop. Separate mesh pointers allow polygonization of only the updated part of the surface, significantly reducing the performance penalty incurred by draw calls to the rendering APIs. Each branch can also signal whether it requires an update or not, depending on whether a sculpt was performed in that region of the grid. This reduces the computational requirements significantly when sculpting on a portion of the surface and allows sculpting in real-time in large grids (e.g. 512^3). With the linear grid implementation, interactive performance cannot be maintained past 160^3.

The octree implementation allows fast voxel searches (e.g. when ray picking) although the neighbor searches are slower than that of the linear grid. Frisken et al. [76] propose methods to allow more efficient neighbor searches which can be used to optimize the octree implementation; although, the implementation

attempts to find neighboring voxels by traversing up through the hierarchy from the leaf node that is querying the neighbors. This can result is a lower number of iterations before finding a neighbor in most cases, but can result in a higher number of iterations in the worst case.

Unlike the linear grid, the octree does not have access to an arbitrary voxel as the said voxel may simply not exist. One therefore has to rely on the narrow band method (see Section 3.3.5) to update the voxels. The minimum size of the narrow band is three voxels long; one intersecting voxel, and 2 neighboring voxels. Updating the narrow band requires extra computations and is thus a performance penalty when compared to the linear grid. In this implementation, the narrow band is expanded to allow the smooth normals algorithm (Section E.2) to function properly. It should be noted that smoothing the normals is a slower operation on the octree, again, because of slower neighbor lookup.

One major advantage that the octree implementation has is the ease of parallelizing all operations. Although in the current implementation a single CPU core is used, all operations on the octree can be parallelized as shown in the first framework (see Section 5.2).

5.4.4 Polygonization Methods

The LSM framework implements both the Marching Cubes and the Marching Tetrahedrons algorithms to polygonize the distance field. The algorithms give slightly different results (Figure 5.7) with Marching Tetrahedrons coping better with curvature changes. The Marching Tetrahedrons algorithm is slightly slower than the Marching Cubes algorithm at mesh polygonization and produces a much denser mesh. The theoretical details of the algorithms are described in more detail in Chapter 3.

Figure 5.7 shows the topological differences between the two algorithms. Note that mesh generation is performed in sections and thus a *sculpt* with the push tool changed the topology around the sculpt and the neighboring voxels (updating the neighboring voxels ensures no tears appear in the surface).

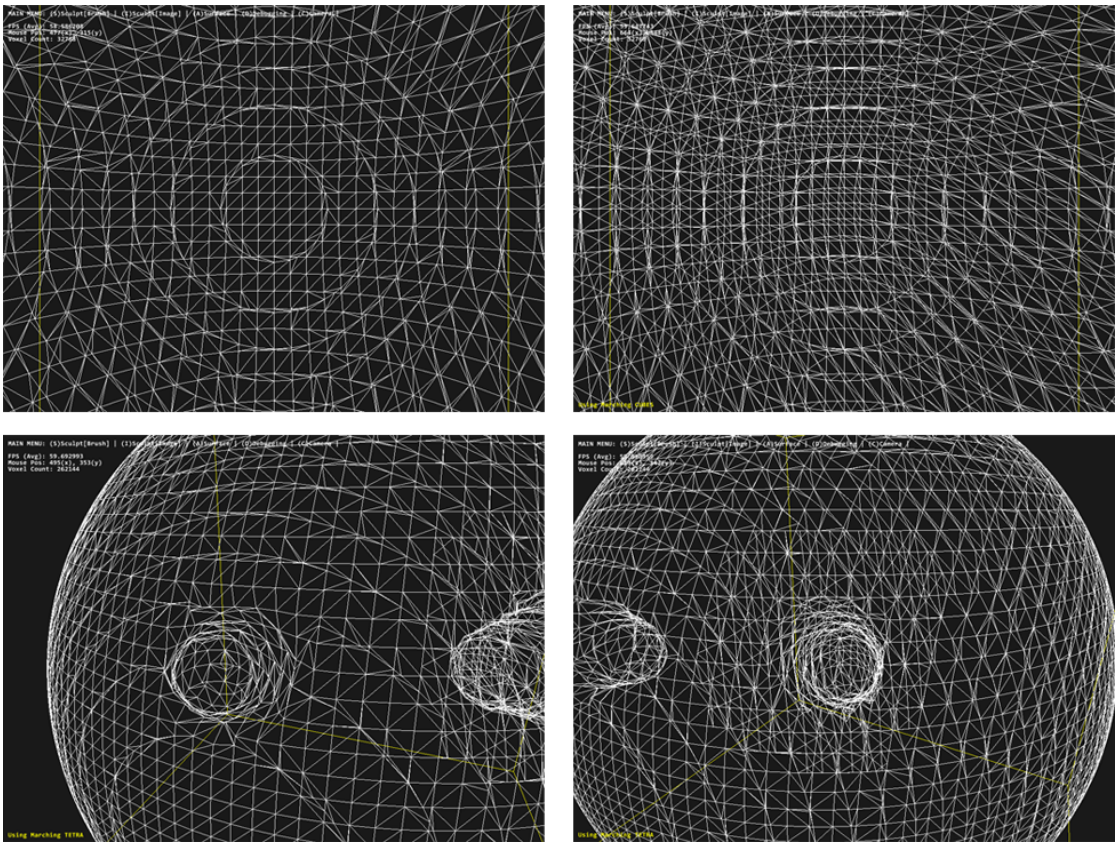


Figure 5.7: Top Left: A sphere visualized through the Marching Cubes algorithm, viewed from the inside, Top Right: The same sphere now visualized by the Marching Tetrahedron algorithm, Bottom Left: A sculpt by using the push tool when using Marching Cubes, Bottom Right: A sculpt using the same tool but this time the affected sections are updated with the Marching Tetrahedron algorithm)

5.4.5 Sculpting by Projection

In this section the sculpting method which uses projection of a depth map on the grid is outlined. Projective sculpting with distance fields has not been explored before for level set methods.

Voxel Selection

The sculpting process is started by selecting all the intersecting voxels in the view frustum of the camera (Figure 5.8(b))

$$V_i \subset V_g \tag{5.1}$$

where V_i are the intersecting voxels and V_g is the voxel grid. Some of the selected voxels may be half visible which is acceptable the depth is sampled accordingly.

The neighboring voxels V_n must now be selected, essentially constructing the narrow band, needed for surface advection (Figure 5.8(c)) such that V_i is the gamma band and V_g is the safe tube and

$$(V_i \cup V_n) \subset V_g \tag{5.2}$$

Finally, any voxels in V_n which are outside the viewing frustum are discarded. A narrow band of voxels is now ready for sculpting.

Voxel Projection

The above selection of voxels can now be projected onto the depth map to approximately determine the pixels (of the depth map) that should be affecting a voxel (see Figure 5.9). All eight corners of the voxel are projected to screen space

$$C_{proj} = M_{proj} \cdot M_{view} \cdot C_{voxel} \tag{5.3}$$

where M_{proj} is the camera's projection matrix, M_{view} is the camera's view matrix, C_{voxel} is a voxel corner $(x, y, z) \in \mathbb{R}^3$ and C_{proj} is the projected corner of the

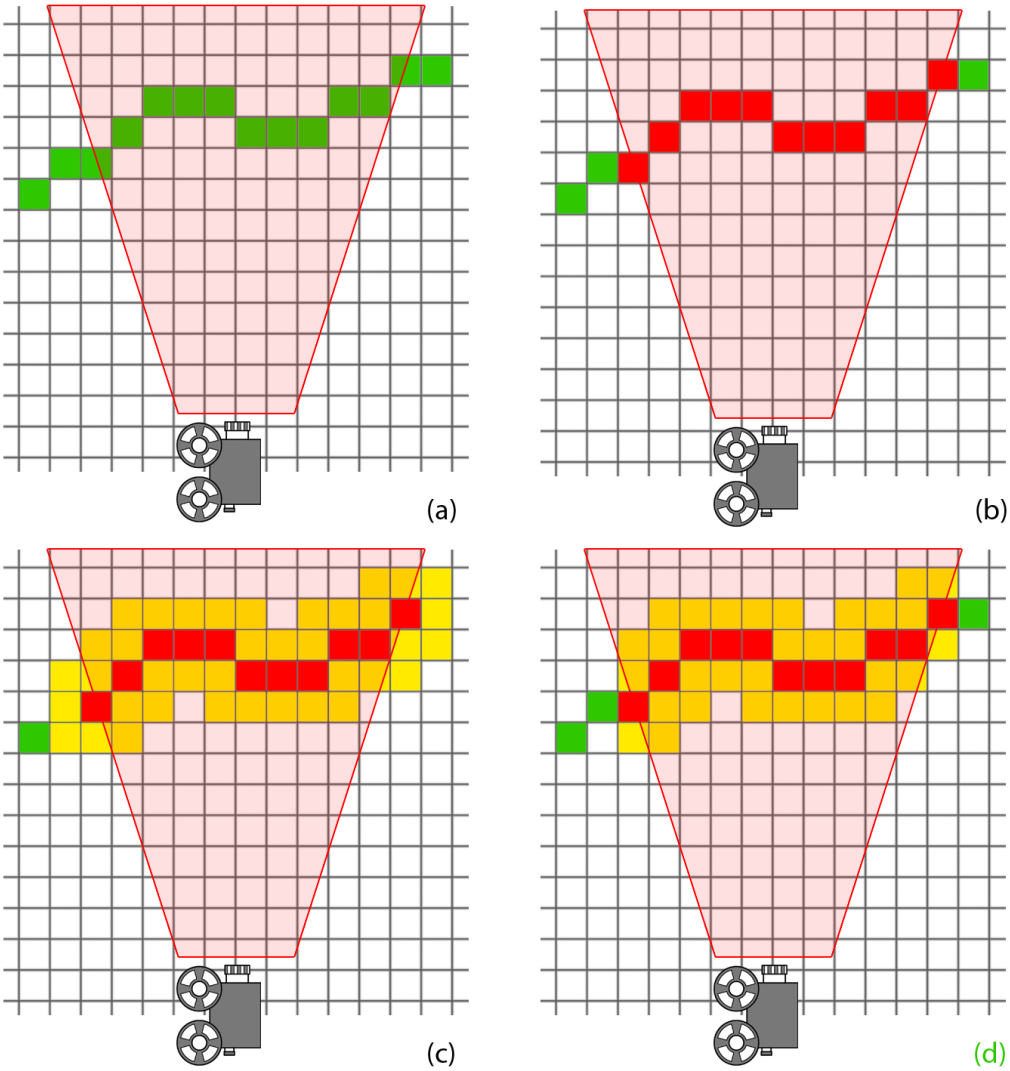


Figure 5.8: (a) The voxel grid with intersecting voxels (shown in green) with the camera and its view frustum (transparent red) (b) The intersecting voxels that are visible by the view frustum (c) The neighbors of all the intersecting voxels that are visible by the view frustum (d) The neighbors that are outside the viewing frustum are dropped

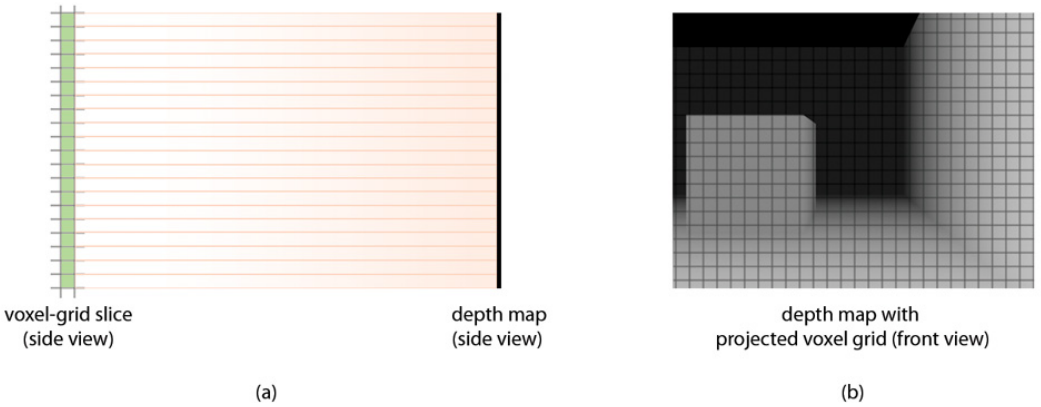


Figure 5.9: (a) Side view showing a slice of intersecting voxels projecting onto a depth map (b) The depth map shown from the front with the projected voxels. Note that a perfect projection is shown for illustration only; generally, the projections are skewed.

voxel $(x, y) \in \mathbb{R}^2$). The left-most and right-most projected corners are selected which are the extents of a square selection. The square is an approximation of the projection of the voxel itself. Any pixels in the depth map falling inside this square are averaged and returned to the sculpting algorithm.

Deprojection

Once the depth value for a particular voxel is calculated, the point needs to be transformed from $\mathbb{R}^2$ back to $\mathbb{R}^3$ or in other words, from view space to eye space. A point p_i in the depth image has an (x, y) co-ordinate and a depth value where $0 \geq x < Image_{width}$ and $0 \geq y < Image_{height}$.

The point p_i must first be normalized so that (x, y) fall between $[-1, 1]$ which is the clip space in OpenGL. This is easily accomplished by

$$x_{clip} = \left(\frac{p_{ix} \cdot 2}{Image_{width}} \right) - 1 \tag{5.4}$$

and

$$y_{clip} = \left(\frac{p_{iy} \cdot 2}{Image_{height}} \right) - 1 \tag{5.5}$$

The returned depth value at co-ordinates (x, y) must also be normalized in a similar fashion so that a co-ordinate (x, y, z) that is now in clip space is obtained.

The next step is to calculate z_{eye} which is the dependent variable for the calculation of x_{eye} and y_{eye}. Calculation of z_{eye} depends on the the characteristics of the camera (virtual or otherwise). This is essentially undoing the perspective divide, where a perspective divide is needed to convert a point from clip space to normalized device co-ordinates.

In this case, renders from Maya give back a linear depth (instead of logarithmic which is usually the case with perspective projections) and the equation using orthographic projection must be derived (see Appendix H for a detailed breakdown of the formulas including their relation to orthographic and perspective projections).

$$z_{eye} = -\frac{(far - near) \cdot z_{clip} + (far + near)}{2} \tag{5.6}$$

where far and $near$ are the clipping planes of the camera. For Microsoft Kinect, depth calibration done by Conley[77] gives

$$z_{eye} = \frac{1}{depth_{raw} * -0.0030711016 + 3.3309495161} \tag{5.7}$$

where z_{eye} is in meters and must be converted to the appropriate units used in the application.

The calculated z_{eye} can be substituted in the following equations to calculate x_{eye} and y_{eye}

$$x_{eye} = -\frac{z_{eye}}{p_{00}} \cdot (x_{clip} + p_{20}) \tag{5.8}$$

where

$$p_{00} = 2 \cdot \frac{near}{right - left} \; and \; p_{20} = \frac{right + left}{right - left} \tag{5.9}$$

and

$$y_{eye} = -\frac{z_{eye}}{p_{11}} \cdot (y_{clip} + p_{21}) \tag{5.10}$$

where

$$p_{11} = 2 \cdot \frac{near}{top - bottom} \; and \; p_{21} = \frac{top + bottom}{top - bottom} \tag{5.11}$$

In the above equations, *top*, *bottom*, $left$ and *right* are the extents of the view frustum of the camera. For a detailed breakdown of how to arrived to the above mentioned equations, please see Appendix H.

The final point $p_{eye} = (x_{eye}, y_{eye}, z_{eye})$ is now in eye space which must be transformed to world space. This is done by multiplying the point with the inverse of the camera's view matrix

$$p_{world} = M_{cam}^{-1} \cdot p_{eye} \tag{5.12}$$

where M_{cam}^{-1} is the inverse of the camera's view matrix. Figure 5.10 shows the steps of the process and the resulting sculpt. The algorithm is listed in Algorithm 1.

Algorithm 1 Projective sculpting

insert all visible intersecting voxels in set V_s
for all voxels in set V_s **do**
  get visible neighbors and insert in set V_s
end for
for all voxels in set V_s **do**
  (x, y) = voxel position projected to screen space
  get depth color D_c at (x, y)
  calculate $(x, y, z)_{eye}$ in eye space
  transform $(x, y, z)_{eye}$ to $(x, y, z)_{world}$
end for
$\delta t_{max} = 0$
for all voxels in set V_s **do**
  calculate gradient G using the upwind scheme
  calculate velocity V from voxel center to $(x, y, z)_{world}$
  if V.length $> t_{max}$ **then**
    $t_{max} = V$.length
    calculate and store H_{xyz} = -(V.dotProduct(G))
  end if
end for
for all voxels in set V_s **do**
  solve the level set equation
end for
update distance field

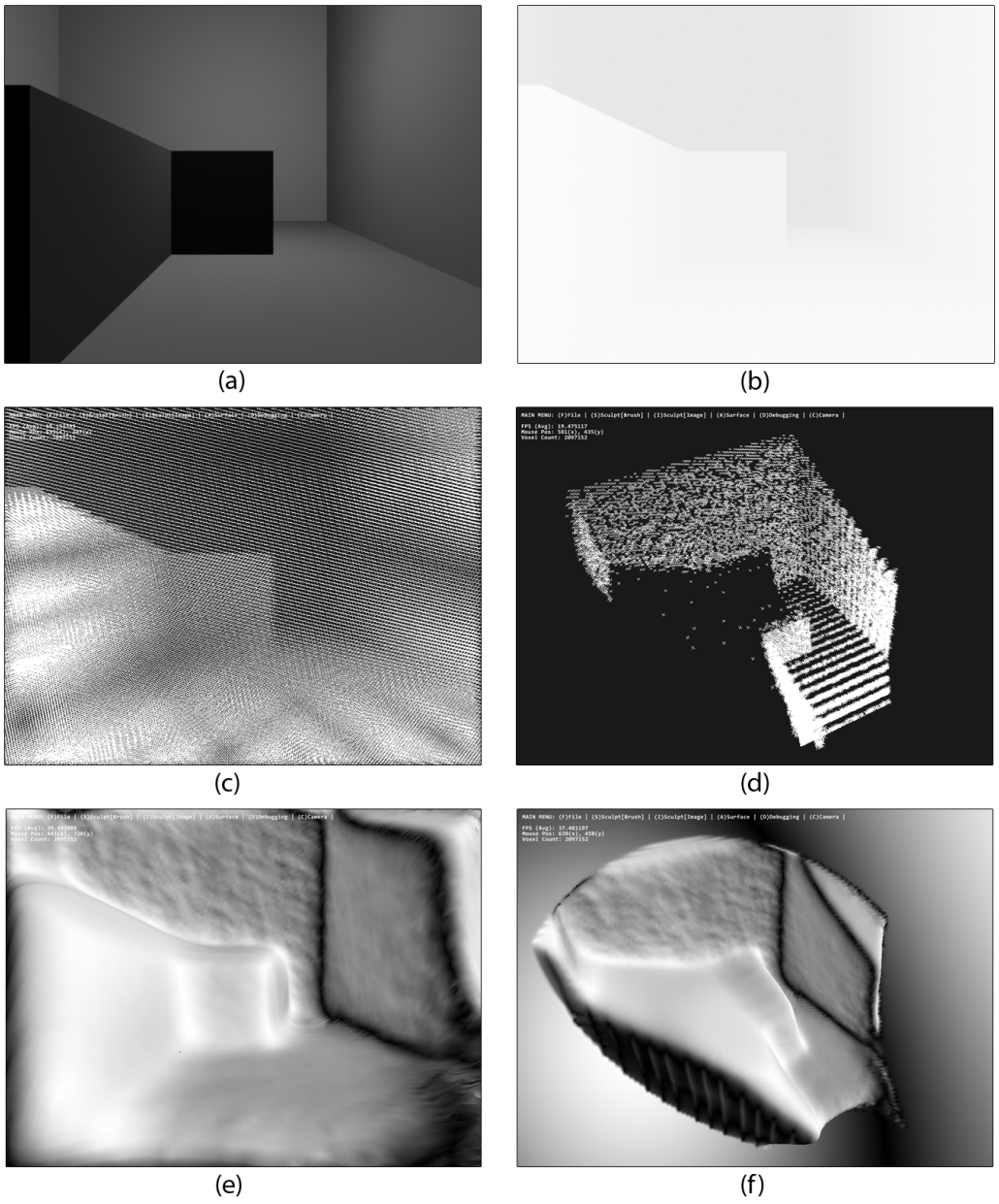


Figure 5.10: (a) The scene rendered from Maya for visualization, (b) The depth render from Maya where the depth is linear rather than logarithmic, (c) The deprojected points, (d) The same points in (c) Shown from a different camera angle, (e) A sculpt performed on an implicit sphere using the methods described in this section, (f) The same sculpt from a different angle.

Chapter 6

Experiments

Three sets of data were collected for use in the frameworks. The first set of data comes from a laser sensor for use in the 2D framework. The second set of data comes from Autodesk Maya (3D Modeling Software) while the third set is from the Kinect sensor. The following sections provide details about the hardware, the experimental setup and the results.

6.1 1D Laser Scanner

6.1.1 Hardware

For the 2D LSM implementation, a laser sensor is placed at different points in a simple environment. The readings were then recorded and used as input in the system. A laser sensor returns range data on a single plane only. For the 3D LSM implementation, a sensor is required that is able to give ranged data in three dimensions; thus, the Kinect sensor was an obvious choice.

The original plan involved a custom built robot with various sections for holding the sensors and a laptop computer receiving readings from the Kinect camera (Appendix F). Due to limitations of the motion capture system (the viewing volume) as well as the Kinect camera (Section 6.3) the Kinect camera was mounted on a tripod. Since the focus of the thesis is on the representation of maps in SLAM algorithm, the assumption is that ground truth data is available and thus this new setup did not pose a problem. The readings for ground truth came from the motion capture setup (see Figure 6.1).

Figure 6.1: The test area surrounded by motion capture cameras. The Kinect camera can be seen on the far right (see Figure 6.12 for a close-up photograph of the Kinect mounted on a tripod)

6.1.2 2D LSM Framework Results

The 2D deformation results were very promising. In Figure 6.2, the red circles denote the locations on the grid where the sensor was placed and the green lines denote the laser measurements at the point denoted by the green grid circle. The distance field is first initialized with a circle of fixed radius, thus the scalar distance stored in each grid cell is initially the distance to the surface of the circle (Figure 4.3(a)). Figure 6.2 shows the entire mapping process in more detail.

For the purpose of demonstrating the process, the SLAM process is shown as alternating between robot motion, prediction and measurement deformation. After incorporating measurements from multiple grid locations, the final map can be found in Figure 6.3(p). Note that since the sensor was always placed in the same orientation, measurements on the far left of the environment were not received; even so, the surface has deformed to accommodate neighbor smoothness constraints. Also note that the zero level set values are set to light blue/green yet do not appear at the resolution of the images in the paper due to sub-sampling. The final contour is continuous and surrounds the deformed area.

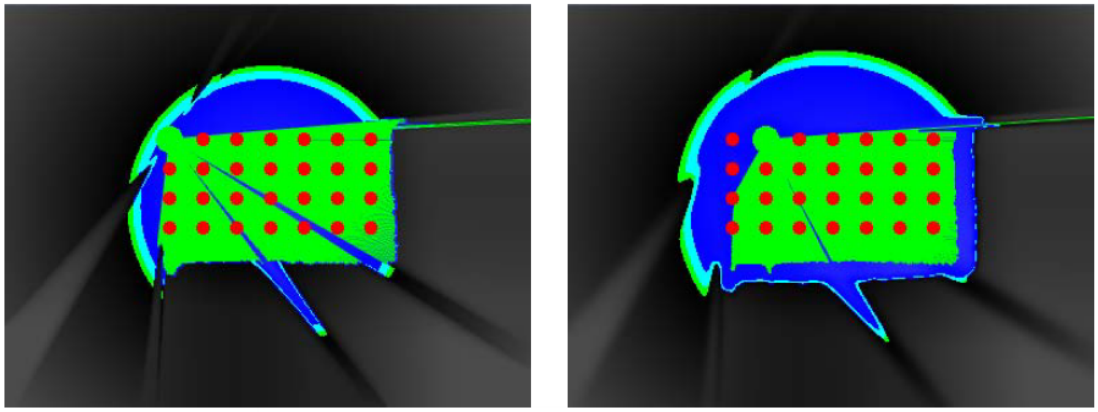


Figure 6.2: Robot Motion as Deformation. As the vehicle moves forward, the uncertainty increases thus the distance field is deformed with a velocity that expands the surface away from the vehicle using the control input to determine how each grid point is changing relative to the vehicle. (a) The map before the robot moves, (b) after the robot moves to the next grid location the map has deformed to accommodate that uncertainty. Gray regions are outside of the surface (positive distance to the surface), blue represents the interior of the object (negative distance) green with aqua colored regions represent the zero level set and the green lines represent the laser line scan data. The red dots represent the locations in which the sensor was placed in the world and the green dot represents the current location of the sensor for this set of measurements. The distortion behind the laser sensor (top and left of the green dot) is caused by the level set solver which in the 2D framework solves for the entire grid even when sensor measurements do not exist for certain parts of the grid.

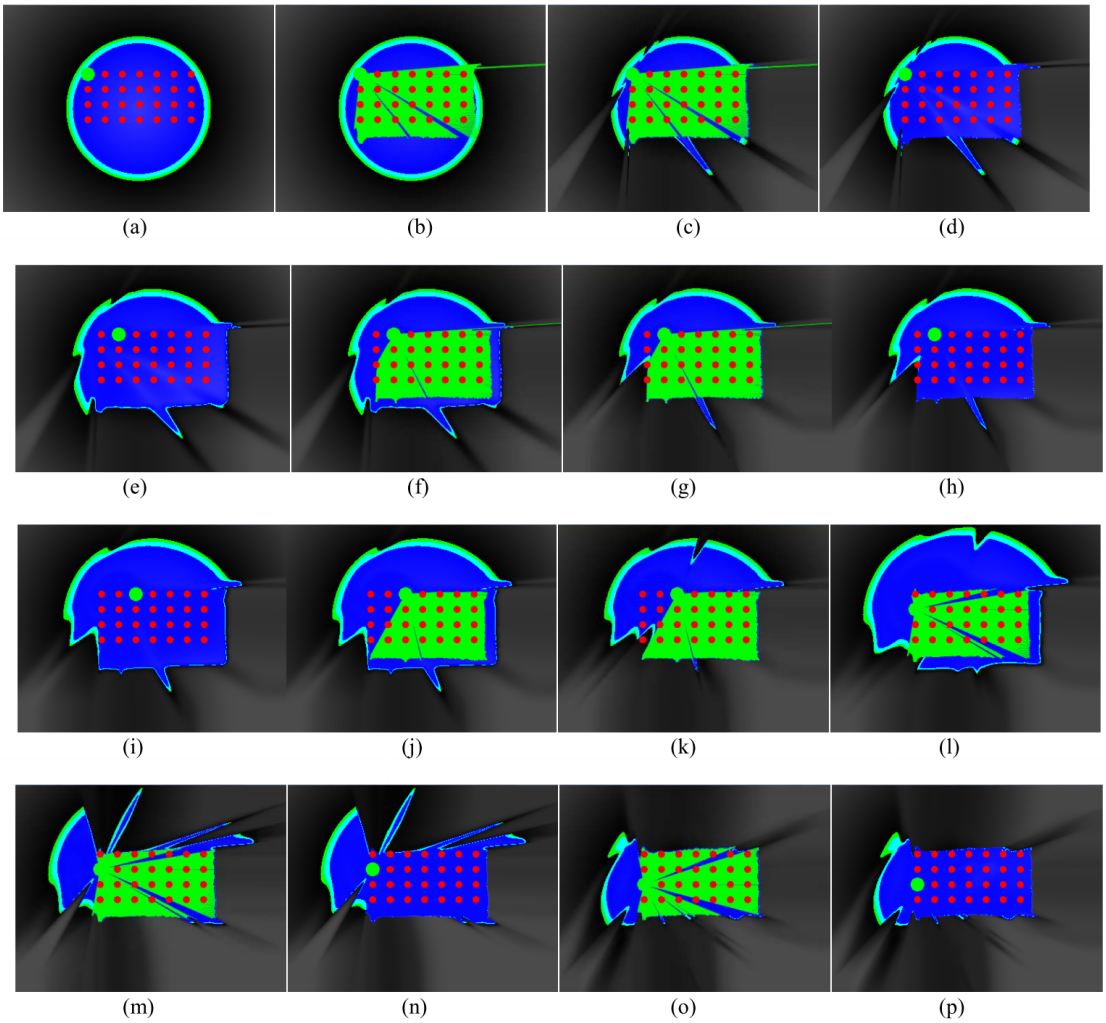


Figure 6.3: Level Set Mapping Process. (a) initial distance field, (b) first laser scan visualized, (c) deformation of the level set after convergence using the sensor measurement field, (d) shows the current map after deformation without the lasers, (e) the sensor moves to the indicated grid location, (f) the results of deforming the map based on the robot motion field, (g) after deforming based on measurements at this location, (h) same as (g) without lasers, (i) sensor moves to next grid location, (j) after deforming by robot motion field, (k) after deforming based on measurements, (l) sensor moved to a different location and after deforming based on robot motion, (m) after deformation based on measurements, (n) without lasers shown, (o) after measurements from entire grid location, (p) the final map. Gray regions are outside of the surface (positive distance to the surface), blue represents the interior of the object (negative distance) green with aqua colored regions represent the zero level set and the green lines represent the laser line scan data. The red dots represent the locations in which the sensor was placed in the world and the green dot represents the current location of the sensor for this set of measurements.

To demonstrate the approach in 2D, results of an experiment are provided using a Hokyo URG-04LX-UG01 scanning range finder (Figure 4.3(d)). The experimental setup can be seen in Figure 4.3(c) as a small enclosed space in the lab. Tables were overturned and placed next to each other to provide rectangular set of obstacles. The laser scanner was placed at 1 foot intervals on a grid throughout the space at the same orientation and the laser range data was recorded.

6.2 Maya Depth Render

6.2.1 Readings from a 3D Scene

The first set of 3D data was collected from a scene modeled and rendered in Maya, a 3D modeling package from Autodesk. The data is essentially *perfect* (i.e. accurate ground truth and depth data with very low error or noise) and gives a base line to which real-world results can be compared. From the Maya scene, a series of depth maps were rendered, similar to that taken from a Kinect sensor, and recorded the camera transformations with each depth render.

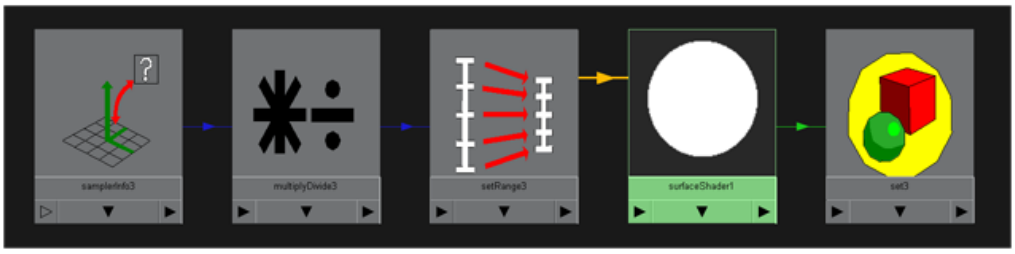
The camera responsible for projecting the depth on the surface was calibrated to that of the Maya camera. In this case, the near and far planes and the FOV of the cameras were matched. Figure 5.4 shows a depth render from the scene.

Before setting up the scene, the camera in Maya must be calibrated to have the same settings as the one in the framework's engine (or vice versa). Maya's camera settings have some caveats that should be noted:

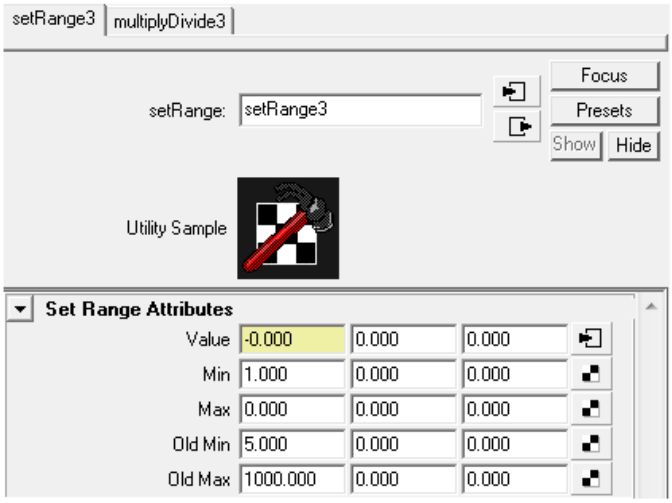
1. Maya has a horizontal Field of View (FOV) (see Figure 6.5 for details about FOV) for their perspective cameras whereas in most 3D engines the FOV is vertical. This was a source of confusion as the attribute was not labeled properly and thus the depth readings were offset.

2. Maya's depth render is linear from 0-255 instead of (the more common) logarithmic. Thus, depth readings are compressed from near/far planes to 0-1 in a linear fashion. Generally, logarithmic depth is used to allow for greater depth precision near the camera.

3. Maya's built-in depth render shader does not use the current camera's near and far planes. This causes the depth render to have a different scale than what is expected (Figure 5.4) with incorrect deprojection. This can be fixed



(a)



(b)

Figure 6.4: (a) The default surface shader that is created for rendering depth, (b) the attributes for the *setRange* node in the shader tree (top middle) where the connections for *Oldmin* and *Oldmax* have been broken and new values put in (in this case, near and far clipping plane distance which is 5 and 1000 units respectively).

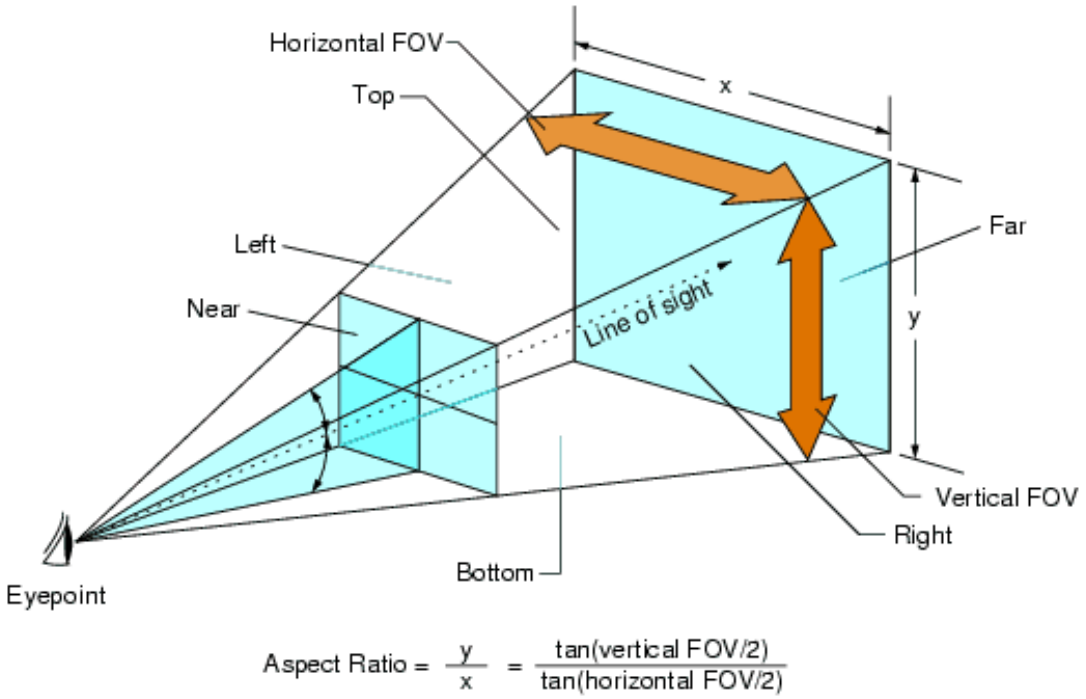


Figure 6.5: Typical camera setup in 3D graphics engines. The camera's view frustum consists of six planes: near, far, top, bottom, left, right. The frustum is usually calculated by a *half angle* (i.e. $\Theta/2$ where the *full angle* is the camera's field of view (FOV). The FOV angle is generally vertical but sometimes it can be horizontal (e.g. Autodesk Maya uses a horizontal FOV) [78].

by breaking the range node connection and putting the correct values (Figure 6.4).

4. Conversion from Maya required the Vertical FOV of Ogre3D to be matched with the horizontal FOV of Maya. This is done by

$$FOV_H = 2 \cdot tan^{-1}(a \cdot tan(\frac{FOV_V}{2}) \tag{6.1}$$

where a is the aspect ratio, FOV_H is the horizontal FOV and FOV_V is the vertical FOV. For steps on how to arrive to this formula, please see Appendix H.

6.2.2 Results

The original Maya scene with all the cameras and the test environment is shown in Figure 6.6. All of the cameras were properly calibrated to match with the camera in the framework. Figure 6.7 shows the six depth renders taken from

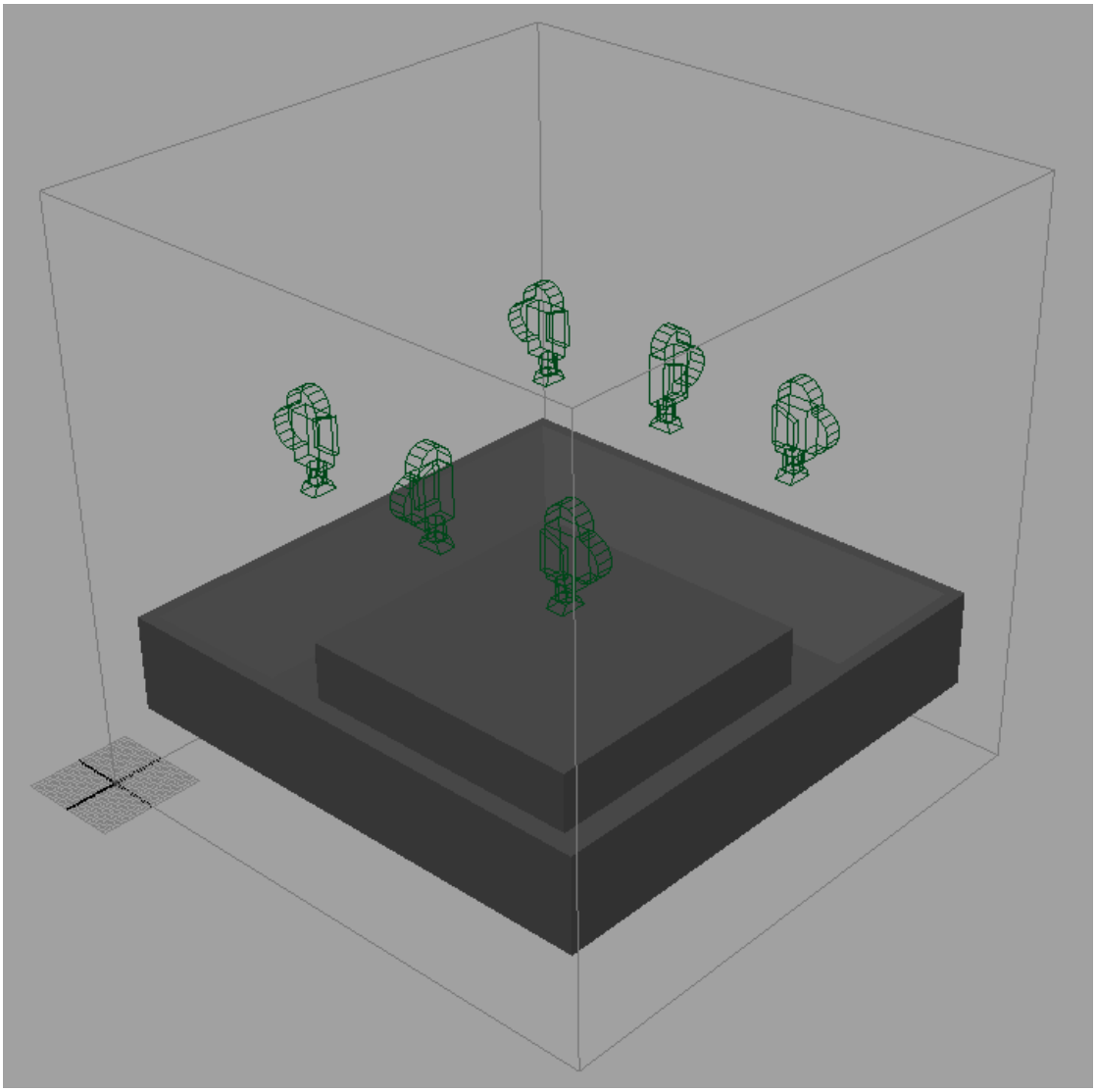


Figure 6.6: The 3D scene created in Maya with six cameras, each responsible for taking a depth render from a certain position and orientation. The grey outline of the cube defines the extents of the setup (which is 128^3).

six different cameras in Maya. The depth image is close to white because of the distance to the scene from the cameras (72 units from the floor of the corridor) and the camera's near and far planes which are at 5 and 1000 units respectively. An object 5 units away from the camera will be pure white and an object 1000 units away will be pure black. Unlike other perspective cameras where the depth scales logarithmically resulting in less precision errors closer to the camera, Maya's depth render scales linearly which results in precision errors. The precision errors cause the deprojection of points in 3D space to show a banding effect (Figure 5.10).

Figure 6.8 shows the first sculpting operation from the depth map in Figure 6.7(a) camera angles. In Figure 6.9(d) the implicit sphere can be seen from a distance with a 2D slice shown in the background. The 2D slice shows the distance to the surface of each voxel in a slice, where black denotes that the voxel intersects the surface, blue denotes that the voxel is inside the surface with a maximum distance of 128, green denotes that the voxel is outside the surface with a maximum distance 128. In Figure 6.9(a) a 2D slice of the original sphere can be seen while Figure 6.9(b) shows the new contour in relation to the deformation shown previously.

Figure 6.10(a) shows the second sculpt as seen from the sculpting camera. Generally, the boundary voxels of the current view and the voxels just outside the boundary have a strong disparity between their distances resulting in a surface tear (Figure 6.10(b)(c)). This can be easily fixed by applying a smoothing filter with a very low threshold (Figure 6.10(d)). Although artefacts can be seen (mainly due to noise in Maya's depth render, angle of the projection on some voxels and precision loss) the resulting surface is stable and continuous.

Figure 6.11 shows the final result. Due to a combination of precision errors, render noise and different convergence rates seams can be seen at each successive sculpt. Notice that depth information at the center of the scene is unavailable where a sharp depression in the surface can be observed. The surface itself, however, remains continuous and free of any holes.

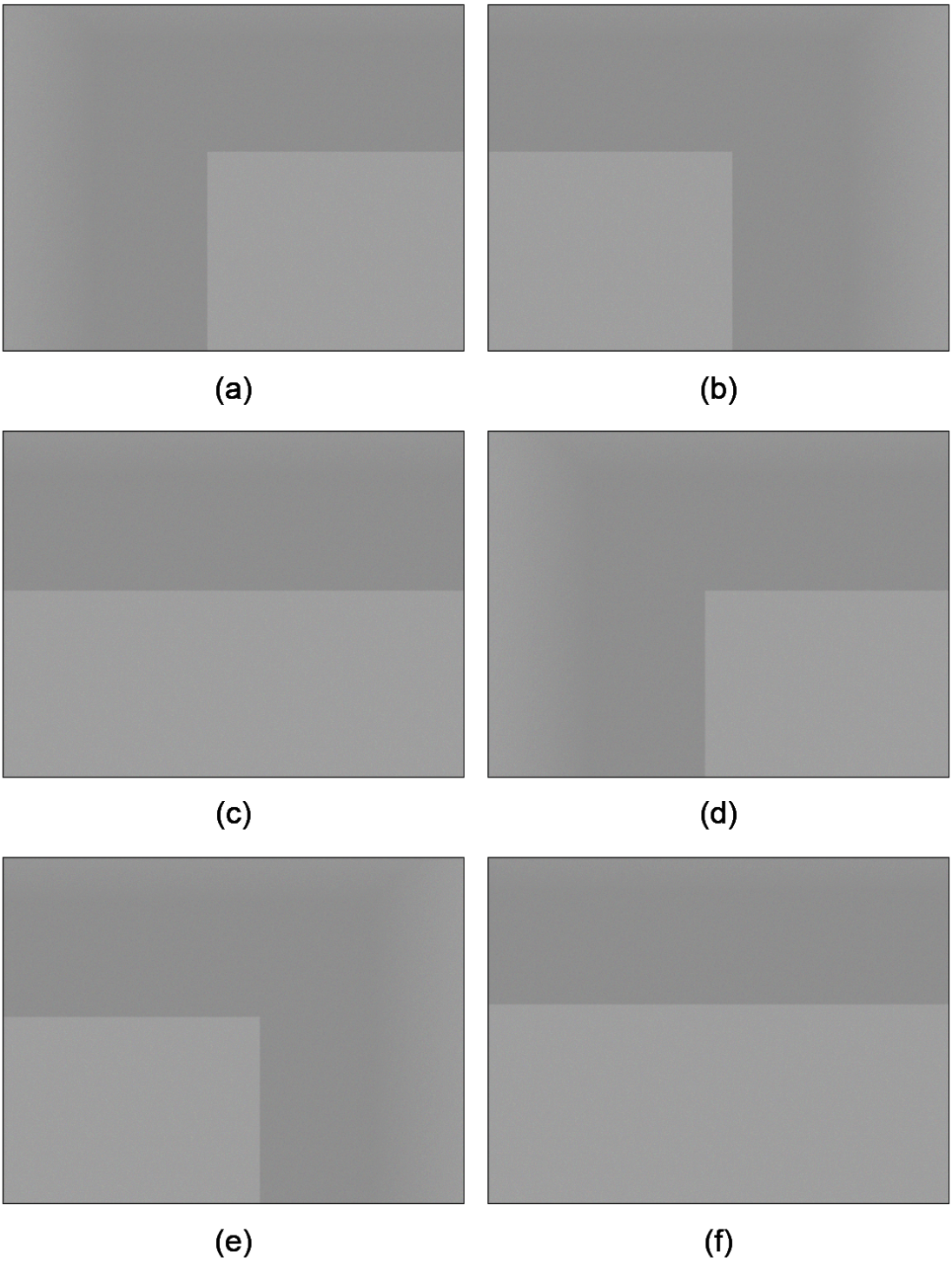


Figure 6.7: (a)-(f) depth renders taken from Maya of a 3D scene shown in Figure 6.8. The brightness and contrast has been adjusted for visualization purposes.

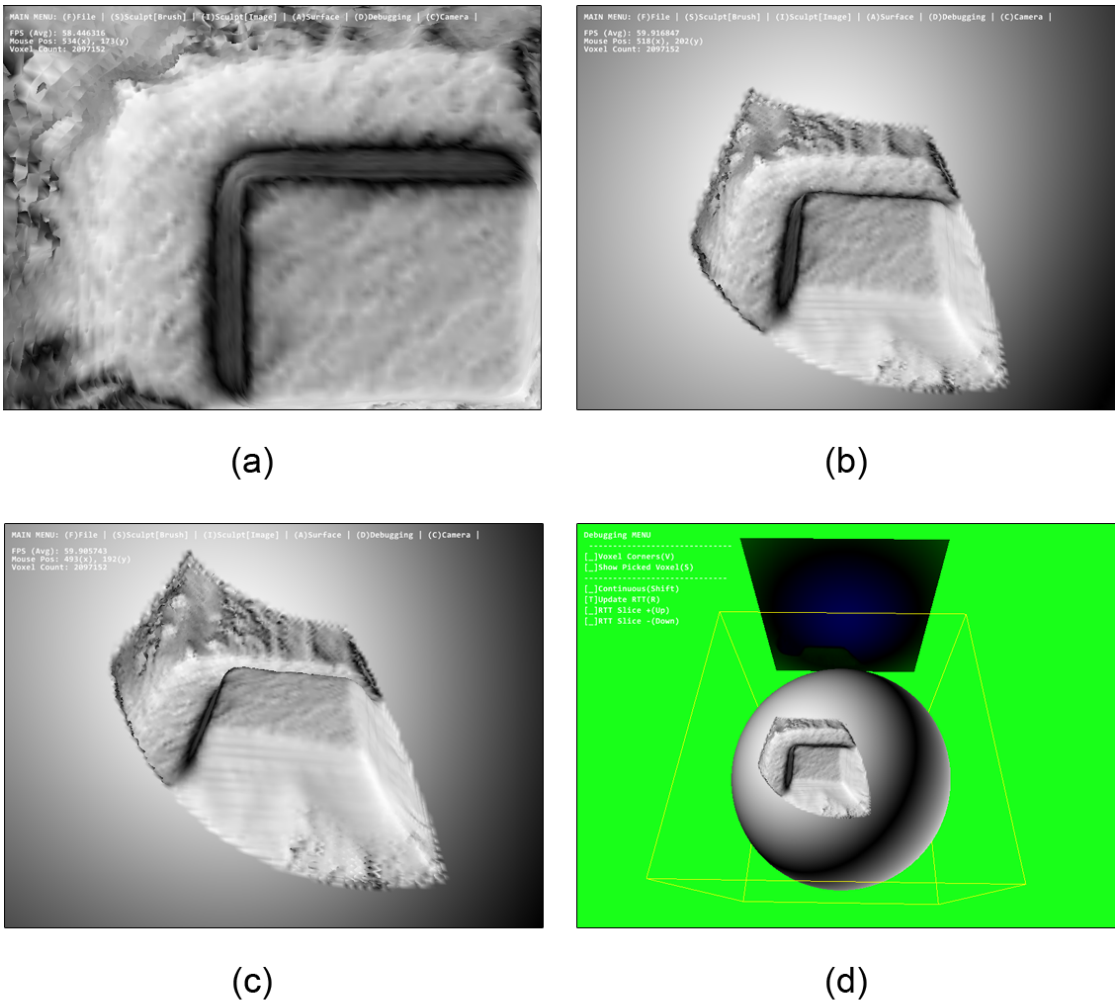


Figure 6.8: (a) The converged sculpt as seen from the sculpting camera, (b)(c) the same sculpt from different camera angles and, (d) the sculpt shown with the implicit sphere (inverted) in full view with a 2D slice shown in the background.

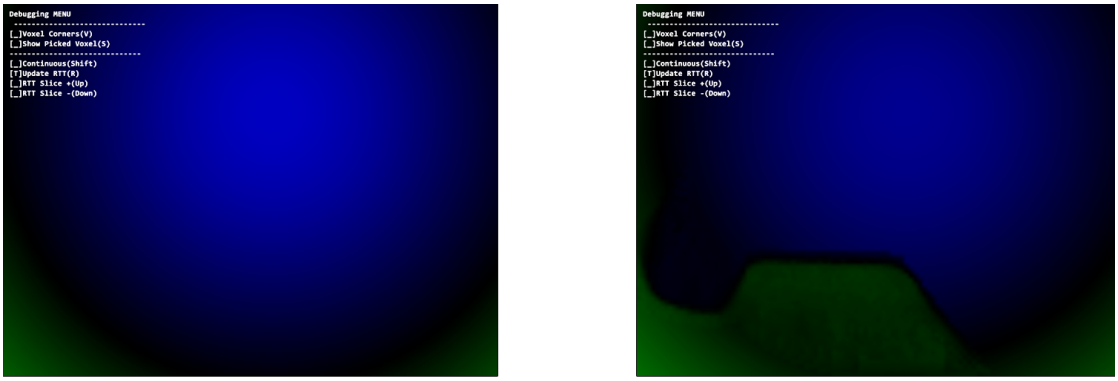


Figure 6.9: (a) 2D slice of the distance field without the sculpt, (b) 2D slice of the same uniuniregion after the sculpt has been performed. Note the change in contour. Black color denotes the voxel is intersecting the surface, blue color denotes the voxel is inside the surface and green colour denotes the voxel is outside the surface.

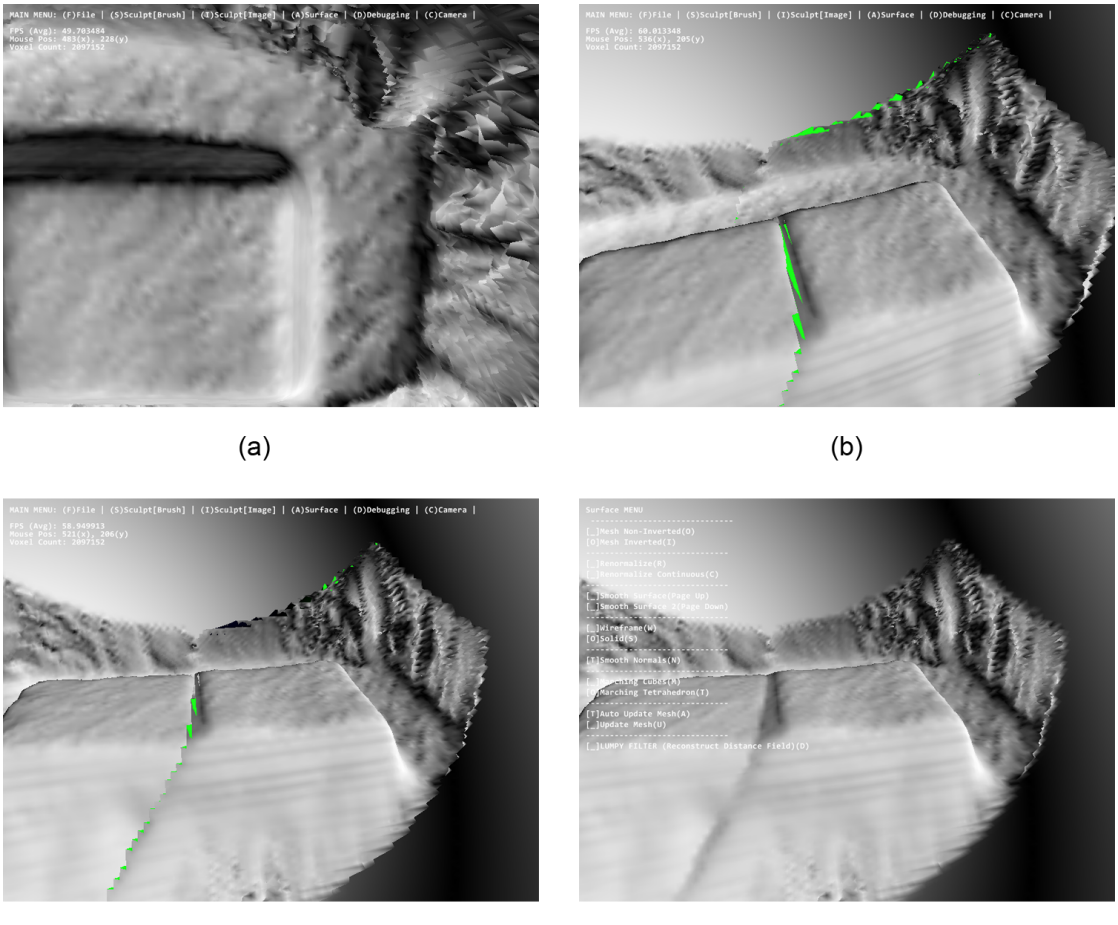


Figure 6.10: (a) the second sculpt as seen from the sculpt camera, (b)(c) different camera angles of the same sculpt showing the holes (green background shows through) and, (d) the repaired distance field with a single smoothing pass.

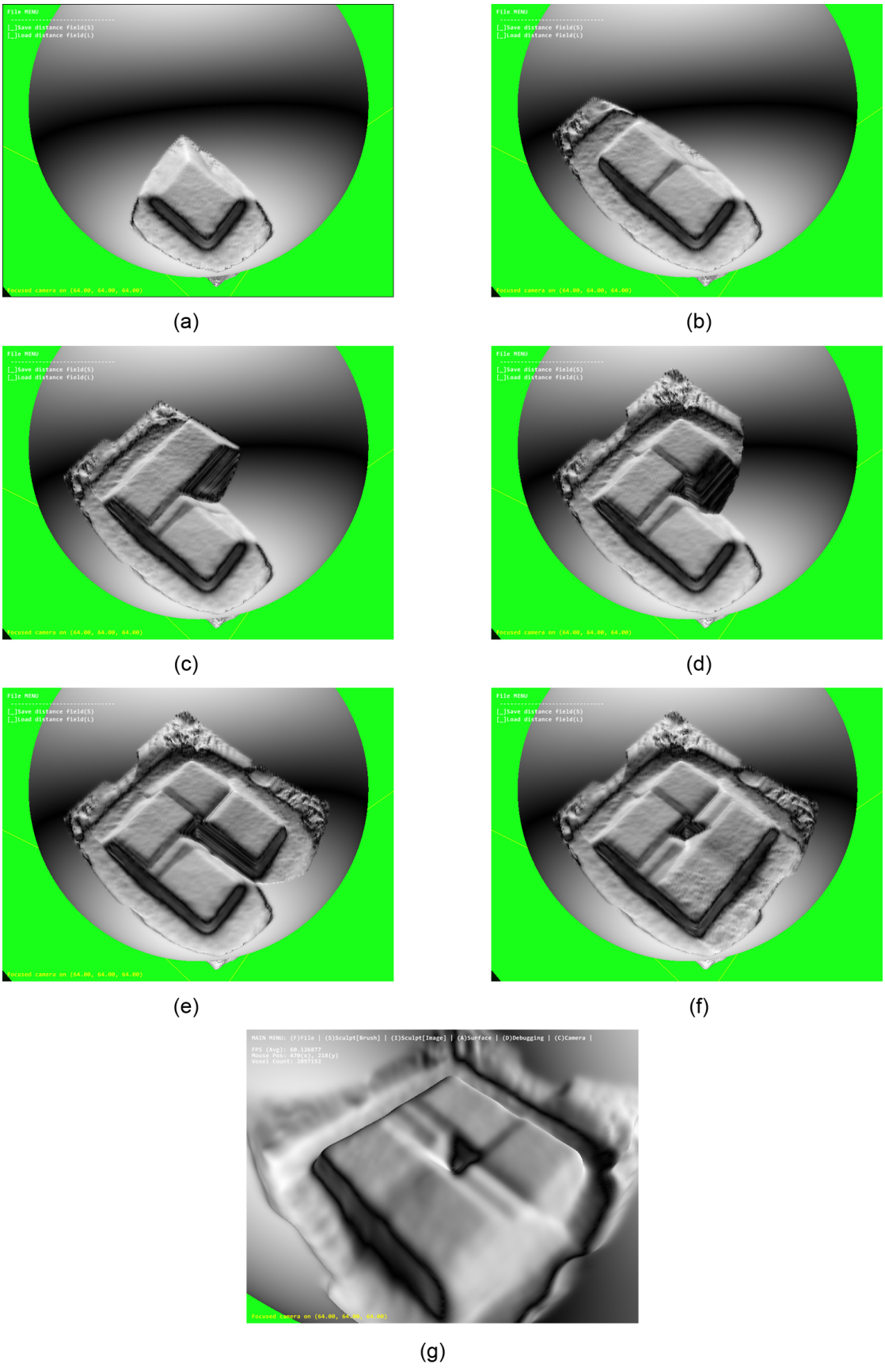


Figure 6.11: (a)-(f) Sculpts from all six cameras from the 3D scene where (f) shows the final result and, (g) the result after a smoothing pass is applied to the complete distance field. The sharp depression in the center is due to the lack of data from the depth renders as none of the six cameras can can see this area. Slight height differences with each sculpt of the box in the center is because of convergence error.

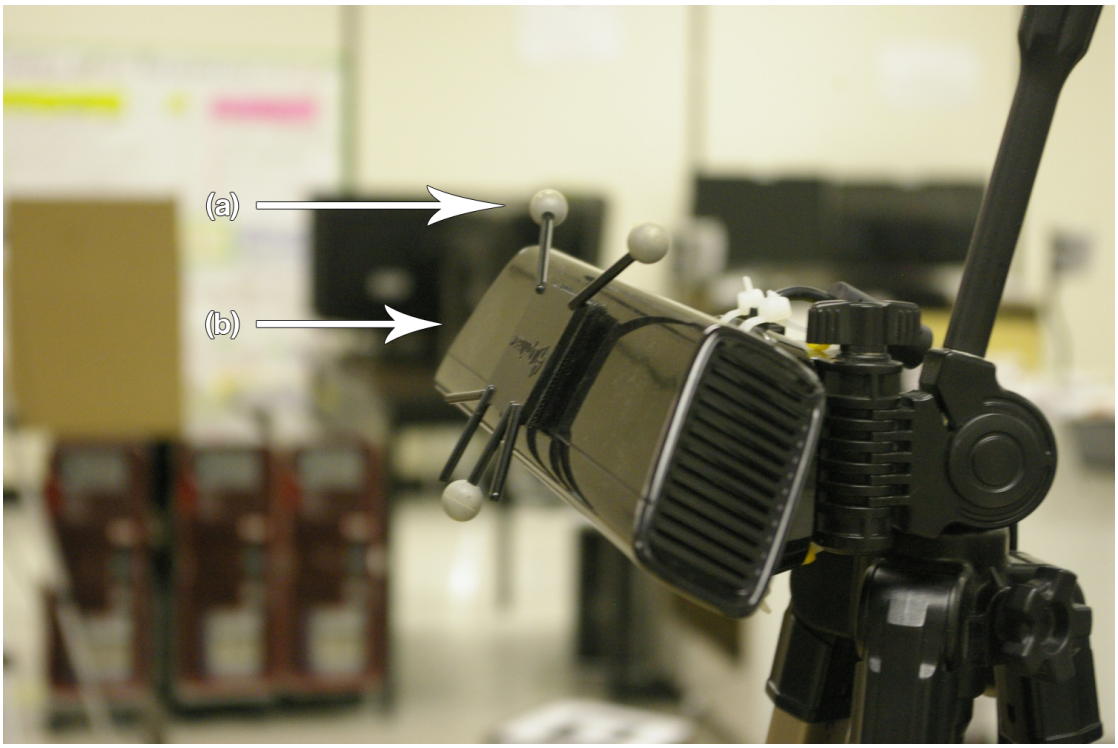


Figure 6.12: (a) One of the three IR markers which the motion capture system can use to construct a plane (b) the Microsoft Kinect camera.

6.3 Kinect Depth Sensor

6.3.1 Hardware

Microsoft Kinect

The Microsoft Kinect sensor (Figure 6.12) is a low cost light based depth sensor that is able to generate depth maps at 30Hz [79]. The resolution of the returned depth is 640×480, however, the images returned are very noisy and with a lot of missing data. The capturing application takes an average of multiple frames to obtain a relatively cleaner and less noisy data for use in the LSM famework (see Figure 6.13). The Kinect camera does not work well when viewing planar surfaces at a very sharp angle (such as the floor when the Kinect is mounted on a robot with low height).

Figure 6.13: A processed depth map taken from Kinect. The capturing application takes an average of several frames to reduce noise. In this case, the depth is an average of 30 frames. Kinect returns values between 0-2047 which are scaled down to 0-255 for use in the framework.

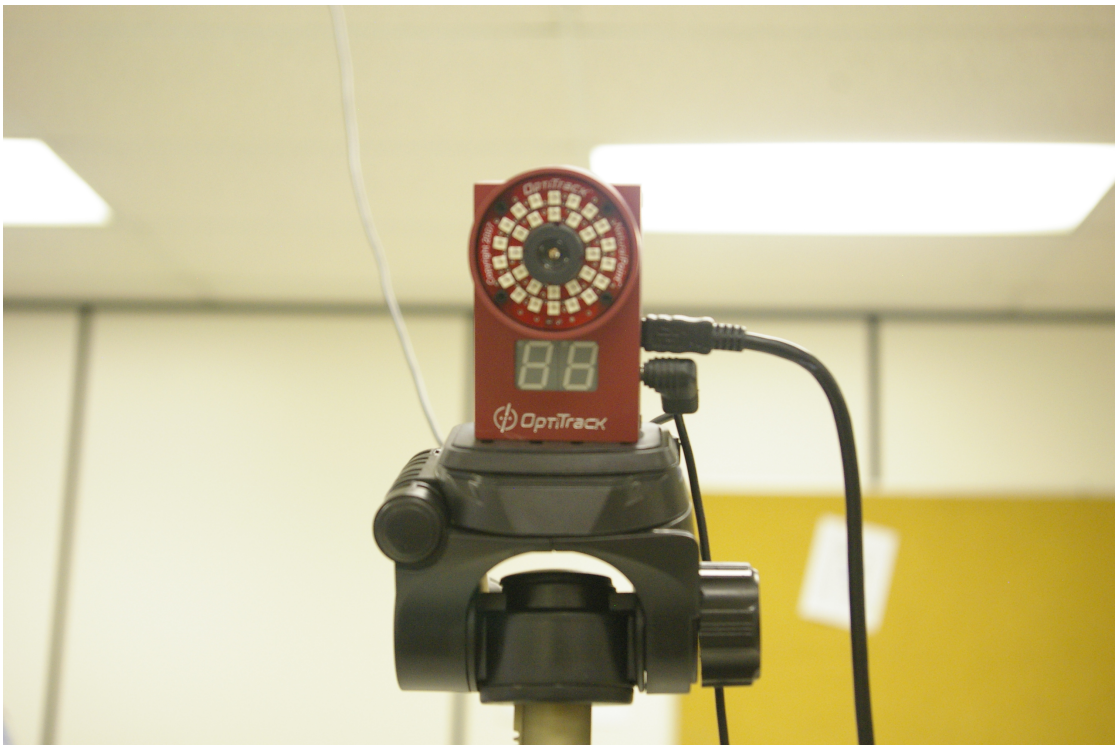


Figure 6.14: An IR camera from the motion capture system (from OptiTrack). The sensor is surrounded by infra-red LEDs set in a circular fashion. The numbered display below the sensor denotes the camera's number when motion capture is active.

Motion Capture

The motion capture system is equipped with eight cameras capable of localizing specially made infra-red retro-reflective markers. Figure 6.14 shows one of the cameras in the system.

For the sensor data, an area of 10$\dot{6}$ feet is tracked by surrounding the area with motion capture cameras mounted on tripods. After calibrating the motion capture setup (Appendix G), the motion of the Kinect camera, mounted on a tripod, can be successfully tracked. The motion capture system is able to track position as well as the orientation of the camera relative to a user-defined origin.

6.3.2 Readings from the Kinect Sensor and Motion Capture System

The Kinect camera was equipped with special infrared reflecting balls which the motion capture cameras used to track the position and orientation of the camera (Figure F.3). There is always some noise associated with sensors and this system

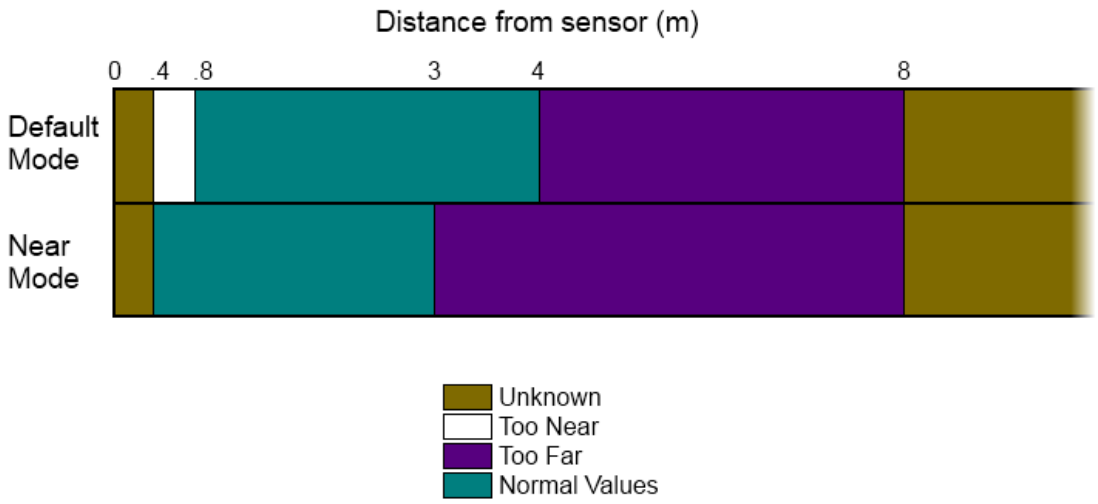


Figure 6.15: The near and default modes of the Kinect sensor. Notice that even in the Near Mode, the sensor cannot detect depth below 0.4 meters or 1.3 feet. Depth information in the purple area is usually too noisy to be useful [80].

is no different. There is a small amount of noise in the positional readings while the orientation data from the motion capture system has a much larger error.

The Kinect sensor has two depth modes that can be selected depending on the location of the scene that one wishes to capture. The near depth mode works upto 10 feet and can take depth readings as close as 1.3 feet. Obstacles closer than the minimum appear black and cannot be discerned in the readings. The far mode can work up to 13 feet but fails to capture any depth for the first 2.5 feet (Figure 6.15).

The depth capturing application stores a series of successive frames recorded directly from the Kinect camera. This results in a less noisy depth image. Color information can be stored, which can be used as textures on a successful surface deformation.

The IR[1] cameras which are responsible for the motion capture were setup in a circular fashion around a 12×12 feet area where the depth readings were taken. Figure 6.1 shows the test area with IR cameras surrounding the area in a (loose) circle. The cameras are adjusted and calibrated to ensure that the IR markers on the Kinect can be viewed at all times within the capture volume (see Appendix G for details on calibration).

[1] infrared

6.3.3 Environment Setup

The aim was to collect data from a depth sensor mounted on a robot. Unfortunately, because of limitations of the Kinect camera (see the following section) and missing information from the motion capture system due to the height of the robot useful data could not be collected for the experiment. The solution to the problem was to mount the camera on a tripod allowing the motion capture system to track the camera properly and reduce sharp angles to allow the Kinect camera to return less noisy depth data.

Figure 6.16 shows the experimental setup with desks acting as the perimeter around the environment with some sections *walled* off with obstacles. Reflective objects interfere with motion capture cameras and dark colors (the original floor) interfere with the Kinect sensor. Therefore reflective materials and darker colors were avoided in the setup (hence the plywood floor). The beveled edges of the desks reflected IR light from the cameras and had to be hidden with a drop-cloth (Figure 6.16).

6.4 Results

Figure 6.17 shows three different depth images from the Kinect camera taken at different locations in the environment. Unlike Maya cameras, the Kinect camera performs some internal processing on the data to ensure that the returned depth is in eye space with a range of $[0, 4000]$. One caveat is that the returned depth also has player information which can be discarded by a bit-shift operation

```
correctDepth = depthFromSDK >> NUI_IMAGE_PLAYER_INDEX_SHIFT;
```

where *NUI_IMAGE_PLAYER_INDEX_SHIFT* is a pre-defined macro in the SDK and can be found in the file *NuiImageCamera.h*.

Figure 6.18 shows the resulting sculpts from the previously mentioned depth images. Notice the relatively severe (as compared to sculpts from Maya depth renders) banding effect which is a discretization artefact and is a known issue [81]. Although the banding effect is not pleasing aesthetically (because of the vertex normals), the combined sculpts are continuous and free of any holes.

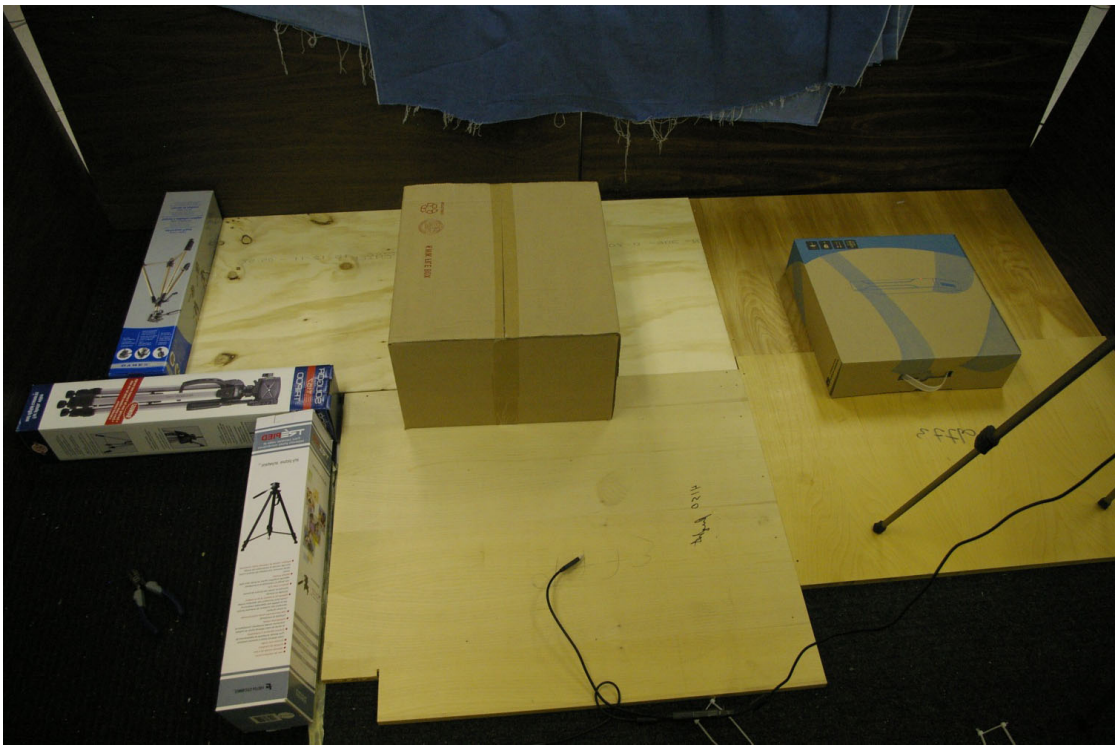


Figure 6.16: Experiment setup with the Kinect camera. A black floor (which had initially been prepared) interferes with the Kinect's depth readings; there the floor was covered with plywood boards. The blue drop-cloth was draped over parts of the upturned desks as the beveled edges were reflecting enough light to interfere with the motion capture cameras.

Figure 6.17: Depth maps from Kinect of the setup where top is the first . Note that the depth is not smooth throughout the range but has a banding effect.

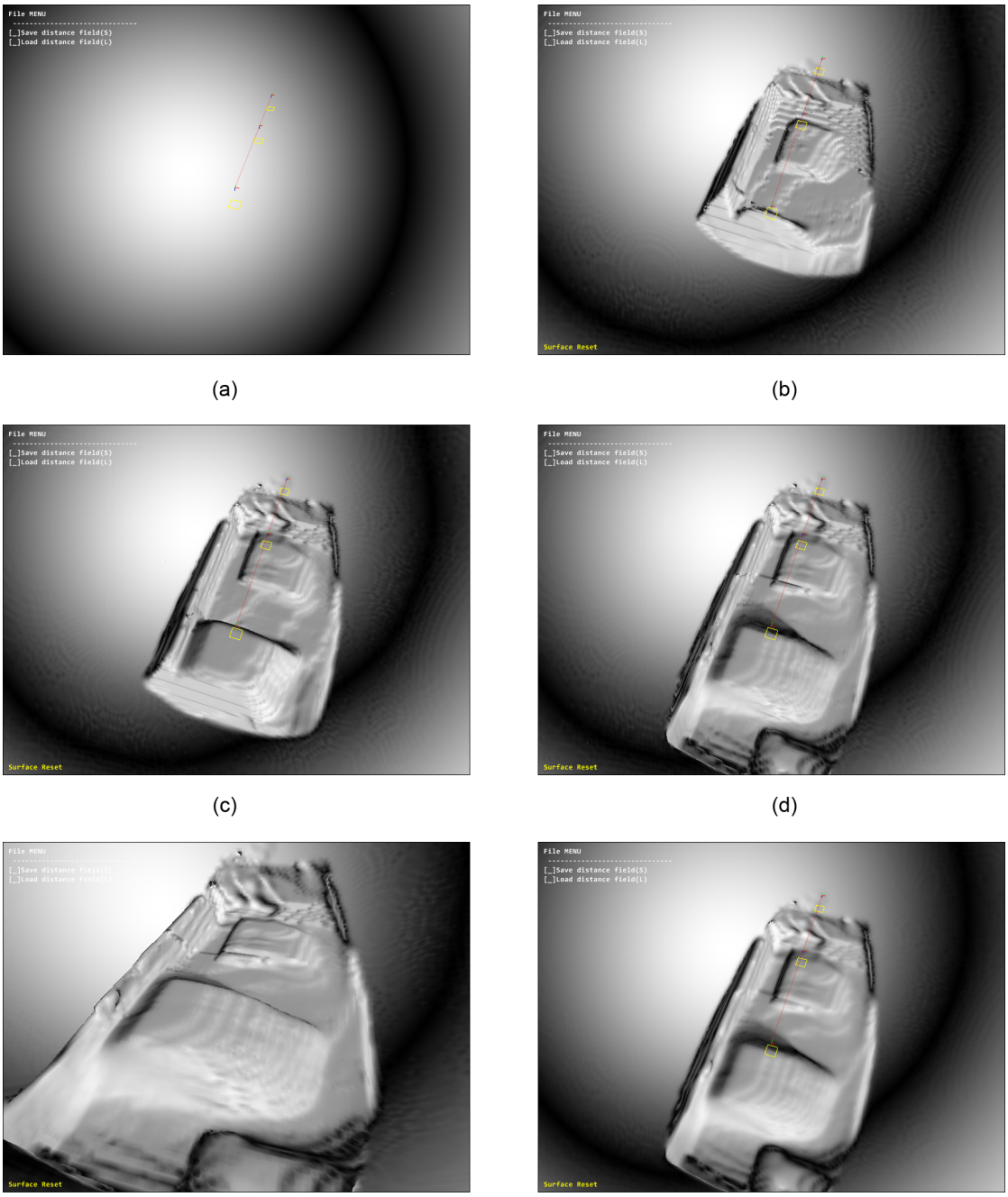


Figure 6.18: (a) Shows the camera positions (red, green and blue lines which represent the pivot point) and the motion path (transparent red line), (b)-(d) successive sculpts from all three locations (each sculpt has a smoothing pass applied after convergence to repair any holes), (e) the final sculpt and, (f) a smoothed version of the final sculpt.

Chapter 7

Discussion & Conclusions

The approach in this thesis explored the use of a level set methods (LSMs) as an alternative map representation for SLAM. Through the implementation map representation with LSMs is not only possible but has several advantages over other mapping techniques such as feature based maps. The implementations developed in this thesis show that manipulation of distance fields through the LSMs can be fast, memory efficient and produce stable results.

If the disadvantages (high computational and memory requirements) are addressed, it would seem that representing the environment with distance fields (which in turn represent an implicit surface) is a perfect fit for SLAM algorithms and will improve the quality of the generated maps. Off-line surface reconstruction algorithms can be applied to further improve and correct the map by filling holes and interpolating with a pre-defined map.

Of course the computational and memory penalties imposed by distance field grids cannot be ignored. While the memory requirements are large due to the need to represent a fine grid, they are constant for a given size of the world that is to be mapped. The memory requirements can be reduced further by using methods such as the hierarchical RLE [72], which is shown to handle high amounts of detail with reasonable memory requirements, adaptively sampled distance fields [38] and other such data structures designed specifically to store sparse level sets. In the framework, an octree spatial tree is used to reduce the memory footprint of the distance field grid.

Mitigating the computational requirements is more challenging since an optimization problem must be solved over the entire grid for each prediction and each

measurement correction stage. In the LSM frameworks the use of multi-threaded libraries and specialized data structures is shown to allow real-time 3D deformation of level sets with relatively low memory requirements (with spatial data structures) on grids as large as 128^3 and interactive deformation on grids as large as 256^3. Graphics processing units can be used to reduce the computational requirements greatly by using the Open Computing Language (OpenCL) framework [82]. The storage mechanism can be improved by integrating ADFs [38], quadtrees/octrees [74] to compress the distance field and then use the narrow band method [73] to reduce the computational requirements even further.

7.0.1 Advantages over Standard Bayes Filters

The advantages of using the methods outlined in this thesis are plentiful. While Bayes filters provide optimal results (for linear Gaussian systems), using Level Set methods as the basis for representing uncertainty enables the use of more complex sensor models and complex dynamics for prediction while maintaining an estimate of the map and its certainty. Spatially varying uncertainty can be incorporated directly into the sensor and robot motion models and the sensor model can be defined as an arbitrarily complex function. A variety of error sources can now be taken into account as long as a velocity field can be defined over an area/volume of space.

Level sets can represent arbitrarily complex geometry which is difficult to achieve in feature based approaches. Color and surface normal information can be stored directly into the representation at the expense of memory, however, this enables the generation of highly detailed models of the environment. Dynamic objects can be incorporated and tracked as long as they are identified first; then, they simply need to be defined as a local minimum in the distance function. Dynamic objects can be treated as yet another velocity field that operates in particular locations of the grid.

This representation also lends itself nicely to navigation and planning algorithms as the world is already defined in terms of occupancy and uncertainty and can be sampled at any location in the Cartesian grid. Finally, one can define and track natural processes that are nearly impossible to represent using feature-based SLAM as long as its evolution can be described by a set of partial differential equations.

7.1 Limitations and Future Work

The implementations have limitations that will be addressed in the future. Currently, because of the voxel selection technique in projective deformation, deformations at very sharp angles (e.g. a wall close to the camera's left or right) corrupt the distance field in such a way that a second deformation pass does not produce expected results.

The applications so far have been compiled as 32-bit binaries. This puts a limitation to the maximum amount of memory that can be utilized. For larger grids, many optimization techniques require more memory. The current LSM sculpting is not multi-threaded and runs slower on most modern multi-core processors than older single core processors such as the Pentium 4 series. This is because of much higher clock speeds on the older processors. With multi-threading, interactively sculpting on larger grids than what was shown in this thesis can be expected.

For the level set solver, Euler integration is used which can accumulate error over time which will be propagated across a sculpting operation. A better more stable approximation can be achieved by the Runge-Kutta method [83], which is a 4th order approximation and may reduce the error in sculpting operations.

The octree data structure is a simpler implementation with slower neighbor lookups. Improvements to the data structure in future versions will allow sculpting with larger brush sizes and images. Currently no color information is stored when sculpting. In future versions texture information can be stored for producing highly detailed environment maps.

One of the bigger hurdles throughout the development of the frameworks had been visualization and debugging the distance fields. Visualizing distance fields (even grids as small as 16^3) is a daunting task as there are simply too much information to sift through. There are very few visualization tools available for level set methods. Seeing the end result is not as informative for the programmer for a particular manipulation. Therefore, to understand the evolution of the distance field between each successive integration a visualization tool is necessary. For future work, a set of generic visualization tools on top of the level set framework is planned.

All LSM implementations suffer from slow deformations because of the *low* maximum Δt that one iteration of advection uses. Even though were able to show real-time deformations, convergence required several iterations due to the way Δt

is calculated (see Section 3.3.1). One solution may be to sculpt the surface in sections separated by *islands* where surface advection is very low. The Δt can then be calculated separately for different sections allowing for faster deformation. This process can also be made to be concurrent.

In future experiments the quantitative error of the resulting error vs the input data set will be evaluated and compared to existing mapping techniques. At this point, however, it was beyond the scope of the thesis.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[2] S. Thrun, "Robotic mapping: A survey," in *Exploring Artificial Intelligence in the New Millenium* (G. Lakemeyer and B. Nebel, eds.), Morgan Kaufmann, 2002.

[3] A. Hogue and S. Khattak, "A variational approach to mapping and localization," in *Computer Robot Vision*, pp. 221–227, 2012.

[4] M. Jenkin, A. Hogue, A. German, S. Gill, A. Topol, and S. Wilson, "Modeling underwater structures," *International Journal of Cognitive Informatics and Natural Intelligence*, vol. 2, no. 4, pp. 1–14, 2008.

[5] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Autonomous Robot Vehicles* (I. J. Cox and G. T. Wilfong, eds.), ch. Estimating uncertain spatial relationships in robotics, pp. 167–193, New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[6] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-whyte, and M. Csorba, "A solution to the simultaneous localization and map building (slam) problem," *IEEE Transactions on Robotics and Automation*, vol. 17, pp. 229–241, 2001.

[7] S. Se, D. Lowe, and J. Little, "Vision-based mobile robot localization and mapping using scale-invariant features," in *International Conference on Robotics and Automation*, pp. 2051–2058, 2001.

[8] S. Se and P. Jasiobedzki, "Stereo-vision based 3d modeling for unmanned ground vehicles," 2007.

[9] M. E. Jefferies and W.-K. Yeap, *Robotics and cognitive approaches to spatial mapping*. Berlin London: Springer, 2008.

[10] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[11] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó, "The slam problem: a survey," in *11th International Conference of the Catalan Association for Artificial Intelligence Research and Development*, (Amsterdam, The Netherlands, The Netherlands), pp. 363–371, IOS Press, 2008.

[12] J. Davis, S. Marschner, M. Garr, and M. Levoy, "Filling holes in complex surfaces using volumetric diffusion," in *3D Data Processing Visualization and Transmission*, pp. 428 –441, june 2002.

[13] J. Leonard and P. Newman, "Consistent, convergent, and constant-time slam," in *International Joint Conferences on Artificial Intelligence*, 2003.

[14] P. Jensfelt, D. Kragic, J. Folkesson, and M. Bjorkman, "A framework for vision based bearing only 3d slam," in *International Conference on Robotics and Automation*, pp. 1944 –1950, may 2006.

[15] S. Thrun and Y. Liu, "Multi-robot slam with sparse extended information filters," in *International Symposium on Robotics Research*, Springer, 2003.

[16] S. Thrun, C. Martin, Y. Liu, D. Hahnel, R. Emery-Montemerlo, D. Chakrabarti, and W. Burgard, "A real-time expectation-maximization algorithm for acquiring multiplanar maps of indoor environments with mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 20, pp. 433 – 443, june 2004.

[17] R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainly," *International Journal of Robotics Research*, vol. 5, pp. 56–68, December 1986.

[18] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "Fastslam: A factored solution to the simultaneous localization and mapping problem," in *National Conference on Artificial Intelligence*, pp. 593–598, AAAI, 2002.

[19] S. Thrun, M. Montemerlo, D. Koller, B. Wegbreit, J. Nieto, and E. Nebot, "Fastslam: An efficient solution to the simultaneous localization and mapping problem with unknown data association," *Journal of Machine Learning Research*, 2004.

[20] J. Guivant and E. Nebot, "Optimization of the simultaneous localization and map building algorithm for real time implementation," *IEEE Transactions on Robotics and Automation*, vol. 17, pp. 242–257, 2001.

[21] J. J. Leonard, H. Jacob, and S. Feder, "A computationally efficient method for large-scale concurrent mapping and localization," in *Proceedings of the Ninth International Symposium on Robotics Research*, pp. 169–176, Springer-Verlag, 1999.

[22] S. Thrun and M. Montemerlo, "The GraphSLAM algorithm with applications to large-scale mapping of urban structures," *International Journal on Robotics Research*, vol. 25, no. 5/6, pp. 403–430, 2005.

[23] S. Thrun, Y. Liu, D. Koller, A. Ng, Z. Ghahramani, and H. Durrant-Whyte, "Simultaneous localization and mapping with sparse extended information filters," *International Journal of Robotics Research*, vol. 23, no. 7/8, 2004.

[24] S. Thrun, W. Burgard, and D. Fox, "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3d mapping," in *IEEE International Conference on Robotics and Automation*, vol. 1, pp. 321 –328 vol.1, 2000.

[25] J. Nieto, J. Guivant, E. Nebot, and S. Thrun, "Real time data association for fastslam," in *International Conference on Robotics and Automation*, vol. 1, pp. 412 – 418 vol.1, sept. 2003.

[26] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *International Conference on Artificial Intelligence*, pp. 1151–1156, 2003.

[27] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *IEEE International Conference on Robotics and Automation*, vol. 2, pp. 116 – 121, mar 1985.

[28] A. Elfes and L. Matthies, "Sensor integration for robot navigation: Combining sonar and stereo range data in a grid-based representataion," in *26th IEEE Conference on Decision and Control*, vol. 26, pp. 1802 –1807, dec. 1987.

[29] B. Schiele and J. Crowley, "A comparison of position estimation techniques using occupancy grids," in *International Conference on Robotics and Automation*, pp. 1628 –1634 vol.2, may 1994.

[30] Y. Liu, R. Emery, D. Chakrabarti, W. Burgard, and S. Thrun, "Using EM to learn 3D models with mobile robots," in *Proceedings of the International Conference on Machine Learning*, 2001.

[31] S. Thrun, D. Hahnel, D. Ferguson, M. Montemerlo, R. Triebel, W. Burgard, C. Baker, Z. Omohundro, S. Thayer, and W. Whittaker, "A system for volumetric robotic mapping of abandoned mines," in *IEEE International Conference on Robotics and Automation*, vol. 3, pp. 4270 – 4275 vol.3, sept. 2003.

[32] J. Bloomenthal, "Polygonization of implicit surfaces," *Comput. Aided Geom. Des.*, vol. 5, pp. 341–355, Nov. 1988.

[33] J. Wang, M. M. Oliveira, H. Xie, and A. E. Kaufman, "Surface reconstruction using oriented charges," in *Computer Graphics International*, CGI '05, (Washington, DC, USA), pp. 122–128, IEEE Computer Society, 2005.

[34] J. A. Baerentzen and K. Lyngby, "Manipulation of volumetric solids with applications to sculpting," tech. rep., Tekniske Universitet, 2002.

[35] J. Gomes and O. Faugeras, "Reconciling distance functions and level sets," in *5th IEEE EMBS International Summer School on Biomedical Imaging*, p. 15 pp., june 2002.

[36] B. Houston, M. Wiebe, and C. Batty, "Rle sparse level sets," in *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, (New York, NY, USA), pp. 137–, ACM, 2004.

[37] C.-L. Huang, "Shape-based level set method for image segmentation," in *Ninth International Conference on Hybrid Intelligent Systems*, vol. 1 of *HIS '09*, (Washington, DC, USA), pp. 243–246, IEEE Computer Society, 2009.

[38] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: a general representation of shape for computer graphics," in *27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, (New York, NY, USA), pp. 249–254, ACM Press/Addison-Wesley Publishing Co., 2000.

[39] M. Jones, J. Baerentzen, and M. Sramek, "3d distance fields: a survey of techniques and applications," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, pp. 581 –599, july-aug. 2006.

[40] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang, "A pde-based fast local level set method," *Journal of Computational Physics*, vol. 155, pp. 410–438, Nov. 1999.

[41] S. J. Osher and R. P. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[42] T. McInerney and D. Terzopoulos, "Topologically adaptable snakes," in *Proceedings of the Fifth International Conference on Computer Vision*, ICCV '95, (Washington, DC, USA), pp. 840–, IEEE Computer Society, 1995.

[43] J. Guivant, J. Nieto, F. Masson, and E. Nebot, "Navigation and mapping in large unstructured environments," *The International Journal of Robotics Research*, vol. 23, pp. 4–5, 2004.

[44] M. Sussman, E. Fatemi, P. Smereka, and S. Osher, "An improved level set method for incompressible two-phase flows," *Computers and Fluids*, vol. 27, pp. 663–680, 1997.

[45] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans, "Reconstruction and representation of 3d objects with radial basis functions," in *28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, (New York, NY, USA), pp. 67–76, ACM, 2001.

[46] Y. Ohtake, A. Belyaev, and H.-P. Seidel, "A multi-scale approach to 3d scattered data interpolation with compactly supported basis functions," in *Shape Modeling International*, (Washington, DC, USA), pp. 153–, IEEE Computer Society, 2003.

[47] G. G. Castro and H. Ugail, "Shape morphing of complex geometries using partial differential equations," *Journal of Multimedia*, vol. 2, pp. 15–25, 2007.

[48] M. W. Jones and R. Satherley, "Using distance fields for object representation and rendering," in *Eurographics (UK Chapter)*, pp. 37–44, 2001.

[49] P. Hastreiter and T. Ertl, "Fast and interactive 3d-segmentation of medical volume data," in *Image and Multidimensional Digital Signal Processing*, pp. 41–44, 1998.

[50] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser, "Collision detection for deformable objects," in *Eurographics State-of-the-Art*

Report, pp. 119–139, Eurographics Association, Eurographics Association, 2004.

[51] M. Eyiyurekli and D. Breen, "Interactive free-form level-set surface-editing operators," *Computers & Graphics*, vol. 34, no. 5, pp. 621–638, 2010.

[52] D. F. Rogers, *An introduction to NURBS: with historical perspective*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[53] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, (New York, NY, USA), pp. 163–169, ACM, 1987.

[54] R. M. Pavel Baxa, Vaclav Skala, "An error estimation for isosurfaces," *Computer Graphics*, pp. 202–211, 1997.

[55] H. Müller and M. Wehle, "Visualization of implicit surfaces using adaptive tetrahedrizations," in *Proceedings of the Conference on Scientific Visualization*, DAGSTUHL '97, (Washington, DC, USA), pp. 243–, IEEE Computer Society, 1997.

[56] G. M. Nielson and B. Hamann, "The asymptotic decider: resolving the ambiguity in marching cubes," in *2nd Conference on Visualization*, VIS '91, (Los Alamitos, CA, USA), pp. 83–91, IEEE Computer Society Press, 1991.

[57] E. V. Chernyaev, "Marching cubes 33: Construction of topologically correct isosurfaces," tech. rep., Institute for High Energy Physics, 1995.

[58] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares, "Efficient implementation of marching cubes' cases with topological guarantees," *Journal of Graphics Tools*, vol. 8, p. 2003, 2003.

[59] R. Courant, E. Isaacson, and M. Rees, "On the solution of nonlinear hyperbolic differential equations by finite differences," *Communications on Pure and Applied Mathematics*, vol. 5, pp. 243–255, 1952.

[60] S. Patankar, *Numerical Heat Transfer and Fluid Flow (Hemisphere Series on Computational Methods in Mechanics and Thermal Science)*. Taylor & Francis, 1980.

[61] T. G. K. Jonas Larsen, "An overview of the implementation of level set methods, including the use of the narrow band method." http://cs.au.dk/ tgk/-courses/LevelSets/LevelSet.pdf, December 2005. [Online; accessed 04-July-2012].

[62] Y.-T. Zhang, S. Chen, F. Li, H. Zhao, and C.-W. Shu, "Uniformly accurate discontinuous galerkin fast sweeping methods for eikonal equations," *SIAM Journal of Scientific Computing*, vol. 33, no. 4, pp. 1873–1896, 2011.

[63] J. Polzehl and K. Tabelow, "Adaptive smoothing of digital images: The r package adimpro," *Journal of Statistical Software*, vol. 19, pp. 1–17, 3 2007.

[64] S. K. Park and R. A. Schowengerdt, "Image reconstruction by parametric cubic convolution.," *Computer Vision, Graphics, and Image Processing*, vol. 23, no. 3, pp. 258–272, 1983.

[65] M. V. Nayakkankuppam, Y. Venkatesh, and Y. V. Venkatesh, "Deblurring gaussian blur using a wavelet transform," 1994.

[66] W. M. Badawy and W. G. Aref, "On local heuristics to speed up polygon-polygon intersection tests," in *7th ACM international symposium on Advances in geographic information systems*, GIS '99, (New York, NY, USA), pp. 97–102, ACM, 1999.

[67] S. F. F. Gibson, "Constrained elastic surface nets: Generating smooth surfaces from binary segmented data," in *Medical Image Computation and Computer Assisted Interventions*, 1999.

[68] R. Sagawa and K. Ikeuchi, "Hole filling of a 3d model by flipping signs of a signed distance field in adaptive resolution," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, pp. 686–699, Apr. 2008.

[69] S. F. Gibson, "3d chainmail: a fast algorithm for deforming volumetric objects," in *Symposium on Interactive 3D Graphics*, I3D '97, (New York, NY, USA), pp. 149–ff., ACM, 1997.

[70] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, Dec. 1987.

[71] R. N. Perry and S. F. Frisken, "Kizamu: a system for sculpting digital characters," in *28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, (New York, NY, USA), pp. 47–56, ACM, 2001.

[72] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth, "Hierarchical rle level set: A compact and versatile deformable surface representation," *ACM Transactions on Graphics*, vol. 25, no. 1, pp. 151–175, 2006.

[73] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations," *Journal of Computational Physics*, vol. 79, pp. 12–49, Nov. 1988.

[74] F. Losasso, F. Gibou, and R. Fedkiw, "Simulating water and smoke with an octree data structure," in *ACM SIGGRAPH*, SIGGRAPH '04, (New York, NY, USA), pp. 457–462, ACM, 2004.

[75] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr, "Level set surface editing operators," in *29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, (New York, NY, USA), pp. 330–338, ACM, 2002.

[76] S. F. Frisken and R. N. Perry, "Simple and efficient traversal methods for quadtrees and octrees," *Journal of Graphics Tools*, vol. 7, p. 2002, 2002.

[77] K. Conley, "Kinect calibration." http://www.ros.org/wiki/kinect_node/Calibration, May 2011. [Online; accessed 04-July-2012].

[78] J. Atwood, "Widescreen and fov." http://www.codinghorror.com/blog/2007/08/widescreen-and-fov.html, Aug 2007. [Online; accessed 28-Aug-2012].

[79] S. Fothergill, H. M. Mentis, P. Kohli, and S. Nowozin, "Instructing people for training gestural interactive systems," in *Computer Human Interaction* (J. A. Konstan, E. H. Chi, and K. Höök, eds.), pp. 1737–1746, ACM, 2012.

[80] C. Eisler, "Near mode: What it is (and isn't)." http://blogs.msdn.com/b/kinectforwindows/archive/2012/01/20/near-mode-what-it-is-and-isn-t.aspx, January 2012. [Online; accessed 04-July-2012].

[81] A.-E. Ichim, "Started implementing 3dgss features." http://www.pointclouds.org/blog/tocs/aichim/index.php, December 2011. [Online; accessed 04-July-2012].

[82] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "Interactive deformation and visualization of level set surfaces using graphics hardware," in *IEEE Visualization*, pp. 75–82, 2003.

[83] J. M. V. Verth and L. M. Bishop, *Essential Mathematics for Games and Interactive Applications, Second Edition: A Programmer's Guide*. Morgan Kaufmann, 2008.

[84] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Cache misses prediction for high performance sparse algorithms," in *4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, (London, UK, UK), pp. 224–233, Springer-Verlag, 1998.

[85] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[86] S. H. Ahn, "Opengl projection matrix." http://www.songho.ca/opengl/gl_projectionmatrix.html, 2012. [Online; accessed 04-July-2012].

Appendices

Appendix A

Software Implementation

A.1 Tools and Requirements

The current implementation is built on a custom engine as well as some external dependencies. The compiled binaries are all 32-bit and thus have an upper memory limit that must be adhered to. In future iterations, 64-bit binaries may be compiled to remove the upper limit.

A.1.1 C and C++

The current implementation uses C++ and templates to allow most decisions to be made at compile time. For example, a policy based design is used to select the grid type (linear or spatial) at compile time without any run-time cost associated with the selection.

A.1.2 Ogre3D

The framework relies on the Ogre3D rendering engine. The rendering engine provides an abstraction over OpenGL and DirectX APIs but allows access to low level functionality. The engine allows switching between the two rendering APIs which can actually be problematic when attempting to use features unique to one or the other. Moreover, some OpenGL functionality is unavailable as the same functionality does not exist in DirectX 9.0 (e.g. getting the depth buffer). In this framework, a shader was developed for capturing depth data.

A.1.3 Intel Threading Building Blocks (TBB)

Modern computers are able to process tremendous amounts of information. With the advent of multi-core processors, the processing can now be done in parallel. To take advantage of parallel processing, serialized code must be made concurrent so that it can run independently on separate threads, and if need be, share information via mutual exclusion. This requires special data structures that are able to process information and perform calculations in parallel without blocking, where one or more threads must wait for another to finish processing on shared data. The choice and design of these data structures depend directly on the task at hand.

Even with fast multiple cores, today's processors are humbled when it comes to calculations involving discretized distance fields. With a specialized data structure, however, it is possible to calculate, construct, render and manipulate distance fields even with large grids (see Appendix B).

Concurrent processing comes with its own share of caveats that must be taken into account. As mentioned, threads must wait on shared data before they can get read/write access, assuming the object or function is thread-safe; if not, then the program will inevitably crash because of memory corruption. A thread that is blocked is a wasted resource and if too many threads are blocked, the performance can be worse than if the program is executed on a single thread. Two threads can acquire locks such that they can enter a state of deadlock, which in the worst case, can stall the program. Thread starvation is another major issue where a thread is not able to acquire locks because of a lower priority and may continue to wait for the duration of the program execution.

The above mentioned drawbacks make it much more difficult to develop stable multi-threaded software. The initial implementations did not take into account thread-safety in the graphics context (when creating meshes) which resulted in memory corruption and inevitably a program crash.

Intel TBB alleviates some of the issues by providing feature rich and stable data structures and a task-based, scheduler controlled threading system. The current implementations use Intel TBB to parallelize the calculations. The TBB library is specifically designed for fast and efficient asynchronous threading and provides a set of concurrent containers (see Appendix B for more details) which is used extensively in the solutions.

The TBB library provides an abstraction over native threads by introducing the concept of tasks. Tasks are analogous to logical threads and can be assigned to a physical thread whenever one is available. The internal task scheduler assigns tasks to idle threads automatically which minimizes the possibility of idle processors.

A.1.4 Visual Leak Detector (VLD)

Memory leaks, especially in memory intensive tasks can destabilize the application (especially 32-bit applications) very quickly resulting in a program crash. Thus the use of memory leak tools, such as Visual Leak Detector, is of high importance to make sure that the program remains leak free.

In this implementations, especially the development of the memory manager, VLD is used extensively to detect leaks resulting in a highly stable memory manager (see Appendix D for results).

Appendix B

Data Structures

This section describes various data structures that are used in the implementations, starting from the basic containers to more complex spatial data structures. Various advantages and dis-advantages in terms of memory usage, speed and concurrent access will be discussed as well.

Although many other data structures exist and have been proposed for use in distance field manipulation, the following have been selected with regards to this thesis's focus on three major aspects of working with implicit surfaces: creating, rendering and manipulating implicit surfaces via distance fields.

B.1 Standard Vector Container

Standard vector containers are essentially raw arrays wrapped into a class which allows transparency when it comes to iteration, resizing and manipulation of the elements along with built-in error checking. Vector containers can be as fast as raw arrays in release builds as they allow random access and cache coherency.

Vector containers, similar to raw arrays, guarantee the data to be laid out contiguously in the memory. Contiguous layout helps alleviate cache misses. A cache miss occurs when a CPU fails to read data unavailable in the cache and is forced to request it from the main memory [84]. Contiguous layout also allows random, constant time access of elements thus making it the fastest containers for random access. Lastly, one unique advantage because of the contiguous layout is using pointer arithmetic to access neighboring elements without access to the container itself. This technique is part of the solution which is used for managing available

elements in one of the implementations of the memory manager discussed later in the paper. Vectors are used for the implementation of the linear distance field grid.

One of the advantages of raw arrays and consequently vector containers, in terms of speed, turns into a disadvantage when memory usage is taken into account. Because of the guarantee that the elements will be contiguous, large vector containers with heavy (in terms of memory usage) objects fail to initialize, even if plenty of system memory is available. The total amount of memory taken up by the objects may be far less than the available memory but if there is not enough contiguous memory to house all the objects because of insufficient stack space and/or memory fragments, initialization fails.

Even though vector containers give the impression of being dynamic, every time the container size is increased, a new array is made, old elements copied into the new array and the old array destroyed. This costly operation makes the containers the worst choice as dynamic containers. The issue can be alleviated to an extent by pre-allocating a certain amount of memory which is then subsequently filled up. Shifting of elements or adding/removing elements from the middle of the array are some of the most costly operations and best avoided (although some implementations can use swap-and-pop if order is not important). Although not inherently thread-safe, since vector containers provide random access to every element, different threads can access the different elements as long as they do not access the same element at once, which in turn will require a locking mutex.

B.2 Intel TBB Concurrent Vector Container

Intel TBB's implementation of the vector container allows concurrent lock-free read access of the contained elements and blocking write access. Perhaps the most useful operations available are the concurrent push and pop operations. Thread-safe access to the elements is achieved via specialized accessor objects. Unfortunately, this adds significant overhead and increases latency [85] and is avoided in the implementations.

One major advantage of TBB's implementation over standard vector containers is that re-sizing an array will not render in-use iterators invalid. This property is very useful when one or more threads is iterating over the container while another is adding elements without blocking.

Similar to standard vector containers, TBB's concurrent vectors provide random access to elements (although without thread safe guarantees) and come with the same disadvantages.

B.3 TBB Concurrent Queue

Queues are dynamic containers with objects as elements that contain a pointer to the user object. These objects also contain references to other elements and hence are linked together, similar to linked lists. The elements of a queue are not contiguous and can be fragmented in memory. Queues have the advantage over arrays when it comes to dynamic resizing. Elements can be added and removed without affecting other elements or requiring a new array to be constructed, thus making the operation take constant time. This ability does not come without costs. One particular performance cost is observed when populating the array with elements. In this case, populating a queue with a million voxels was approximately two times slower than populating a vector container.

TBB's queue implementation allows concurrent push and pop operations in a first in last out fashion, working very much like a stack. A useful operation that is leveraged heavily in one of the implementations is a non-blocking pop operation *try_pop*(). This operation attempts to dequeue an element, and will not be blocked if the attempt fails. This allows the thread to continue to perform other tasks.

The major limitation of queues (and lists) in general is their inability to provide random access to elements. To find an element in the queue, one must start iteration from the start, which in the worst case can take O(n) time, where n is the number of elements in the queue. This, coupled with pointer dereferences for access to the next element makes them a poor choice for element iteration and therefore should only be used as a queue, as the name suggests.

B.4 TBB Concurrent Hash map

Hash maps implement an associative array, where a key (hash) is required to access a data element. With good, unique hash keys, a hash map can bring down the cost of insertion, deletion and lookup operations to O(log n). Erasing elements, assuming a reference to the element already exists, takes constant time. These properties will prove to be very useful in one of the implementations.

Hash maps, unlike queues, do suffer from performance hits when inserting elements into the container. This is because of the container trying to insert the element in such a way as to balance the data structure for optimal insertion, deletion and lookup operations. TBB concurrent hash maps are no different and may block other threads, also trying to insert elements, especially if the hash key used is of poor source (e.g. pointer address converted to an unsigned integer).

B.5 Octree Spatial Tree

Octree is a spatial data structure, commonly used in 3D graphics for culling, that is utilized to perform computations on the distance field, which can take advantage of multiple processors to perform lock-free, thread-safe scalable multi-threaded operations. The data structure is able to compress the original grid to store the distance field which reduces its memory footprint while at the same time allows it to perform parallel, non-blocking calculations on the voxels. The octree is able to subdivide selectively (see Figure 6), thus reducing the number of calculations performed on any given node and more importantly, allow each branch and its subsequent child branches or leafs, to perform calculations independently.

The octree performed exceptionally well in earlier tests, with very large grids taking less than a second to compute, polygonize and display when run on a quad-core system (see Figure take figure from final graphics report). Unfortunately, since the data structure is not contiguous in nature, memory fragmentation is the biggest problem encountered in previous attempts, so much so that the program would crash if a second request was made to generate a surface with a grid resolution of 512^3. The octree suffers from a few other disadvantages (such as slower neighbor lookup) which are detrimental to distance field manipulation, especially sculpting and morphing.

The independence enjoyed by the branches and thus the grid's voxels, which has proven very useful for parallelizing the marching algorithms, is also detrimental when it comes to finding voxel neighbors. In the worst case scenario, the lookup will take O(log n) time (starting from the parent, it will take n calls to reach the lowest division level), although grids as large as 2048^3 will take no more than 11 calls to reach the desired voxel. Even so, in practice hundreds of thousands of lookups per frame can contribute to a heavy drop in performance.

The octree's memory consumption can be optimized further and can behave similar to ADFs by deleting (or recycling) the voxels that are found to not intersect the

surface. Consequently, a parent voxel that is intersecting a surface may or may not have 8 equally divided child voxels, thus saving memory and computations. When manipulating the octree, several thousand *new*() and *delete*() calls may be made which fragment the memory. Eventually the system is no longer able to issue enough memory for a single branch containing a voxle as data and thus terminates the program. The issue is examined in more detail and a solution provided in Section memory fragmentation and memory manager section.

B.6 Other Data Structures

There are other spatial data structures that have been shown to be highly useful in generating high resolution distance field grids with low computational and/or memory costs. Some of the notable data structures proposed for level set methods include the ADFs (Adaptively sampled Distance Fields) proposed by Frisken et al. [38] and the RLE sparse level set structure proposed by Houston et al [36].

The ADF data structure is able to represent the surface with sharp edges by only subdividing the required voxels until an approximation of the curvature can be made through trilinear interpolation of the subdivided cells. Since the subdivision is based on the rate of curvature change, there is no preset maximum resolution for an ADF. Thus, an ADF is able to represent a sharp edge at a higher resolution than other techniques while at the same time reducing the memory and computation requirements by storing lower resolution voxels in areas with lower or no changes in curvature. Precise carving of surfaces is now possible without using excessive memory [38].

The ADF data structure does suffer from high coupling between various nodes in the spatial tree. Moreover, generating the tree while taking into account the curvature for every voxel, before determining whether to discard it or not, would incur a computational cost. In the worst case, in a surface with high curvature, the ADF may be no better than its simpler alternative, the Octree. With higher computational costs, it may perform worse. Nevertheless, ADFs have been shown to perform exceptionally well in a deformation framework, called Kizamu, by Frisken et al. [71] where fairly complex triangle model was generated from the distance field in 0.37s on a Pentium IV processor (an outdated processor by today's standards, but still one of the most powerful single core processors compared to today's lower clocked multi-core systems).

Kd trees are specialized BSP trees with properties very similar to the previously discussed Octree. In the current solution, a kd tree is not used, but it is worth mentioning because of its superior subdivision properties. Subsequent iterations of the program will replace the Octree with a kd tree. An Octree has to subdivide a large cell into eight individual cells even if a small portion of the larger cell is intersecting the surface. In a kd tree, the split can be made anywhere, and there is no requirement for the split to divide the parent cell equally. This allows for better compression of the original grid. The splitting technique does increase the design complexity of the tree. Since the position of the split can be moved, each node of the tree requires extra storage. The nodes, unlike an Octree, are not guaranteed to be in the shape of a cube which increases computation requirements of algorithms that rely on intersection tests.

B.7 Compression of Spatial Trees

Houston et al. proposed run length encoding (RLE) to remedy the memory requirements of sparse level sets such as octrees and ADFs [36]. The proposed solution did not adapt to spatial grids. A subsequent iteration of the algorithm, also proposed by Houston et al. is Hierarchical RLE [72]. Although RLE compression is slow for real-time usage, a variation of it can be used to decrease memory usage while minimally impacting performance. Hierarchical RLE is able to compress a 5000 x 3000 x 3000 grid of 45 billion voxels to fit 1GB of memory. To give some perspective, if each cell in the grid stores just one bit and is stored in a traditional linear array, it will take more than 5GB of memory. Clearly, the RLE algorithm warrants more research and perhaps modification to make it feasible for use in real-time applications.

Appendix C

Smart Sub-Select

Neighbor lookups in spatial trees are very costly. If the number of neighbor lookups can be reduced to determine sign changes in a voxel, thousands of costly lookup operations can be dropped. In this section a novel algorithm will be described that *predicts* which corners of the voxel may undergo a sign change and can be used to quickly determine whether the voxel intersects a surface or not.

A voxel needs to be subdivided recursively until the smallest predetermined size is reached or the voxel does not intersect the surface. To determine whether the voxel intersects the surface, all known algorithms determine the distance to the surface on all eight corners of the cube, whether the eight corners a stored within a voxel or calculated by querying the neighboring voxels. Note that this algorithm relies on the intersecting voxel storing or calculating the direction vector to the surface.

In Smart Sub-Select, each voxel calculates the distance and direction to the surface from the center of the voxel. The direction is then used to find the corners that may be result in a sign change in the distance field. The computationally expensive distance to the surface calculation is then performed only on those corners.

To minimize the amount of calculations performed to determine the required corners, a lookup table is used. For example, if the direction to the surface from the center of the voxel is positive in the x-axis, positive in the y-axis and negative in the z-axis, then the only possible corner that may have a sign change is corner #8 of the cube (see Table C.1).

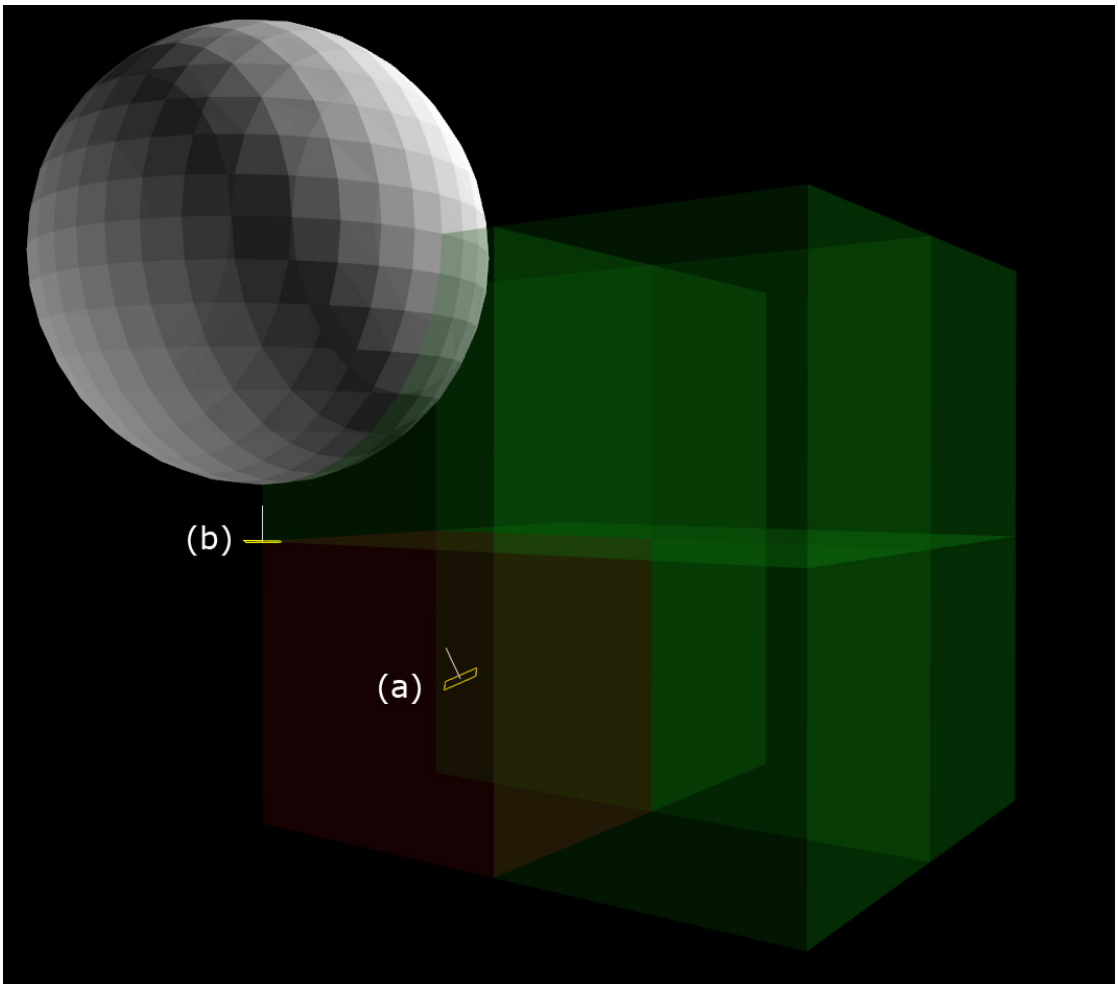


Figure C.1: (a) direction from the center of the voxel (b) the corner tested for a sign change

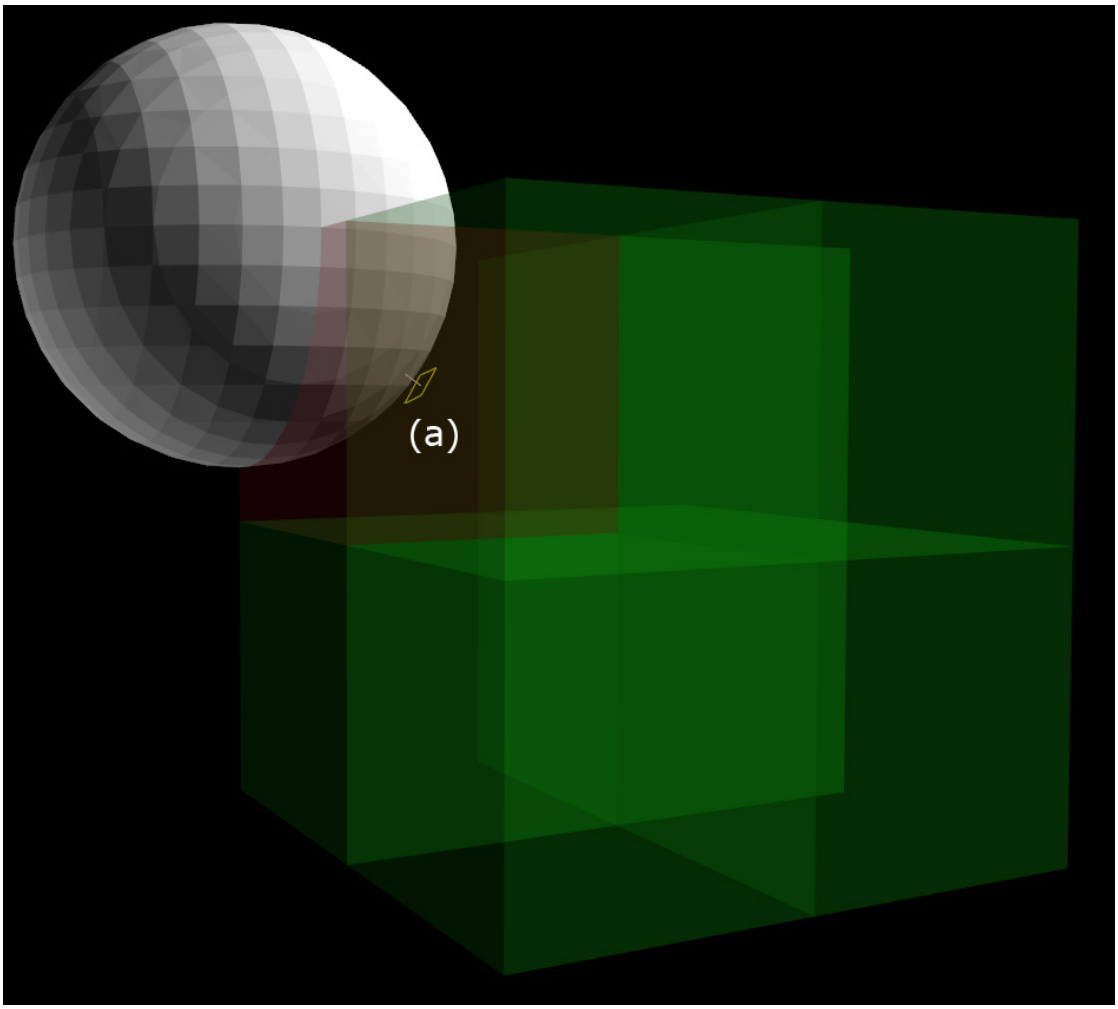


Figure C.2: (a) similar to Figure C.1, only this time the resulting corner undergoes a sign change

X	Y	Z	Corner(s)	Packed Corners	Hex Value
+	+	+	7	(01000000)	$0x40$
+	+	−	8	(10000000)	$0x80$
+	+	0	7,8	(11000000)	$0xC0$
+	−	+	3	(00000100)	$0x04$
+	−	−	3,4	(00001100)	$0x0C$
+	−	0	4	(00001000)	$0x08$
+	0	+	7,3	(01000100)	$0x44$
+	0	−	4,8	(10000100)	$0x88$
+	0	0	4,3,7,8	(11001100)	$0xCC$
−	+	+	6	(00100000)	$0x20$
−	+	−	5	(00010000)	$0x10$
−	+	0	5,6	(00110000)	$0x30$
−	−	+	2	(00000010)	$0x02$
−	−	−	1	(00000001)	$0x01$
−	−	0	1,2	(00000011)	$0x03$
−	0	+	6,2	(00100010)	$0x22$
−	0	−	1,5	(00010001)	$0x11$
−	0	0	1,2,5,6	(00110011)	$0x33$
0	+	+	7,8	(11000000)	$0xC0$
0	+	−	5,8	(10010000)	$0x90$
0	+	0	5,6,7,8	(11110000)	$0xF0$
0	−	+	2,3	(00000110)	$0x06$
0	−	−	1,4	(00001001)	$0x09$
0	−	0	1,2,3,4	(00001111)	$0x0F$
0	0	+	2,3,6,7	(01100110)	$0x66$
0	0	−	1,4,5,8	(10011001)	$0x99$
0	0	0	1,2,3,4,5,6,7,8	(11111111)	$0xFF$

Table C.1: This lookup table can be used to determine the corner that may have a sign change and discard the corners that are guaranteed not to undergo a sign change.

To illustrate this idea further, in Figure C.1 a corner is determined to be likely to have a sign change. Distance calculation is then performed on that corner relative to the surface and it is determined that that the corner does not undergo a sign change relative to the center of the voxel and the voxel is thus discarded. In this case, seven extra computationally expensive calculations were not performed.

In Figure C.2, using the same technique, a corner is determined to undergo a sign change and therefore the voxel is then subdivided further. Unlike other methods, such as the ADFs, these calculations can be performed on each voxel independently.

Appendix D

Memory Manager

Any memory allocator can cause fragmentation with enough allocations and deallocations. An allocation is also a relatively expensive process which can be felt in a program performing thousands of allocations and deallocations per frame. In the first implementation for distance field rendering and manipulation, out of memory exceptions were frequently encountered if too many requests were made.

A general solution is to use a memory pool which is specialized for a particular object. Moreover, with today's multi-core systems, a good memory pool should allow multiple threads to read from the pool lock-free and even write to the pool of objects without stalling any threads.

The following sections describe in detail the problems encountered and the solution which has worked well in the implementations given in this thesis.

D.0.1 Problems

Memory Fragmentation

Memory fragmentation, similar to disk fragmentation, is caused by contiguous chunks of data occupying the memory in a sparse manner, such that larger chunks of contiguous memory are not available; thus, an object that is larger than the largest chunk available, cannot be instantiated, even though the total available system memory is more than enough to accommodate the object (see Figure D.1).

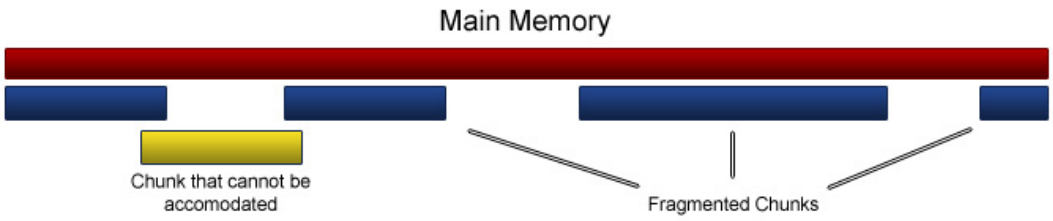


Figure D.1: The red bar represents main memory while the blue bars represent allocated memory. The yellow bar is a new request by an application for a specific amount of contiguous memory. The request for the allocation in this case is denied.

Cache Misses

A cache miss may occur when a CPU is unable to fetch the next instruction from the cache and is forced to access the main memory, which is much slower than the on die memory caches. There can be many reasons for a cache miss; unnecessary pointer de-references, nested function calling, pointer aliasing, fragmented memory, non-contiguous data structures and failed branch predictions [84].

Thread Blocking

Turning a serial code into concurrent code may not necessarily result in an increase in processing speed. Moreover, a program running perfectly serially may cause memory corruption when converted to utilize multiple threads. In an application that must manipulate and process millions of elements per frame, any concurrent access that blocks threads will be detrimental to performance. As an example, trying to *pop_back*() a value from a concurrent queue which is already busy in the same or another blocking operation, will result in the thread stalling and thus wasting frame time. Sometimes multiple threads

Resource Allocation/De-Allocation

Allocating and de-allocating resources on the fly not only causes memory fragmentation, as mentioned earlier, but is also very expensive, especially if the program allocates and de-allocates thousands or millions of objects per frame. Also, the memory chunk of a deleted resource may not be given back to the system, which causes the program to behave as if it is leaking memory. With many allocations, finding the resource leak is exceptionally hard and specialized programs, such as Valgrind, must be used to find the leak; although memory leak detectors slow down

the execution speed by orders of magnitude and may not make them a viable choice for finding memory leaks in complex programs.

D.0.2 Solution

The solution to the above mentioned problems is a dedicated memory manager that manages all the elements that will be used and discarded frequently in large numbers. It should be able to initialize a set number of maximum elements and be able to handle requests that result in more elements than are available by informing the user of such requests. In the context of this project, voxels, triangles and (mathematical) vectors can make good use of a well-designed memory manager.

The initial LSM framework did not use a memory manager. It would instead instantiate the root voxel in the the octree, prepare the distance field, perform the march and create the mesh. Once done, the root voxel would be deleted, which in turn deleted all the children and the meshes. Another run would start with a fresh root voxel and behaved as if the program was just launched. This behaviour prevented distance field manipulations such as morphing between two surfaces for larger grids. Even on smaller grids, the morphing would cause enough memory fragmentation to force the operating system to terminate the process. Thus, one of the main requirements for the new memory manager is for elements to be recycled in any order.

The following solutions, failed or otherwise, all make use of a memory manager class which every object that wished to create new elements of type voxel, Triangle or Vector3f had access to. It is not, however, limited only to these three types. The memory manager class utilizes templates and thus can work with any type.

Initial Attempts

During the development of this project, many attempts were made to select the best combination of data structures and algorithms to achieve the best performance and lowest memory footprint. Listed below are some of the attempts that were made with brief discussion as to why they failed to satisfy the goals.

Concurrent Queues The first attempt made use of two concurrent queues; one queue was for available elements, and then other queue for used elements. The available queue was initialized by pushing all the available voxels into the queue before the start of the program. The in-use queue was empty at the start of the

program. When an element is requested, the next available element is popped from the in-use queue and pushed into the in-use queue.

Before starting this attempt, recycling of random elements was not an option and would have to start fresh simply because queues did not support erasing elements in between the front and back elements. This attempt was a stepping stone towards more elegant solutions.

The use of two queues as described did not change the behavior of starting from scratch every time, but it did manage the memory better than the original program and was able to perform many more calculation runs before running out of memory because of fragmentation.

The program suffered from fragmentation because every time all the in-use elements were recycled, the in-use queue would erase the pointer to the element. Moreover, the results were also approximately twice as slow as the original program which was mostly due to the expensive push operations on the concurrent queue. Because of the recycling limitation, this particular solution could not be used on $Vector3f$.

Concurrent Queue and Vector A method must be devised to recycle random elements at any time during the run. A concurrent queue and a concurrent vector were used to store the free voxels and voxels that are in use respectively. The queue was initialized by pushing all available voxels into the queue before the start of the program. The vector is initially empty.

When an element is required, it is popped from the concurrent queue, inserted into the concurrent vector container and a reference returned to the thread that requested the element. The vector container can then be traversed as required e.g. to create 3D meshes from triangles stored in the container.

This approach allowed recycle of random elements and was a big step towards the right direction. The recycling of elements however was very slow. Each time an element was randomly freed, the concurrent vector must be traversed to search for the newly freed element and push it back onto the queue. It was anticipated that these operations will be performed every frame and this solution was clearly not going to be able to handle even moderately sized distance field grids.

The solution did work better than the previous attempt because this time instead of destroying the pointer pointing to the now used element, the pointer is simply set

to NULL in the concurrent vector container. However, the memory requirements were increased because the concurrent vector container must contain the same number of pointers as the total number of elements.

Proposed Solutions

Below are two solutions that have been implemented in the system. These solutions have their share of advantages and disadvantages. Depending on the requirements, one can be picked over another. It should be noted that the solution with concurrent queues and hash map is slower than the original solution which did not use a memory manager, although it is more reliable with a smaller memory footprint even on very large grid sizes. The concurrent vectors with indexing solution on the other hand is faster, stable and more reliable than the original solution but requires more initial memory although there is no further increase in memory consumption.

Concurrent Queues and Hash Map The last technique with the concurrent queue and concurrent vector was modified and the vector container was replaced with a concurrent hash map. A reliable integer to hash function was chosen to be used with the hash map in an effort to allow the map to balance itself as much as possible. This solution was the first of the two that worked reliably and relatively efficiently while lowering the memory footprint.

The solution was slower by approximately $2.5x$ than the original program including the last attempt. This was because of the expensive push and insert operations on the queue and the hash map and the fact that iterating over the hash map (required to create the final meshes) is slower than a vector container. This solution is be able to handle two distance fields morphing, albeit not in real time for grids larger than 16^3.

In an attempt to decrease the chance of a thread being blocked because of the above operations, functionality was added to the memory manager to allow adjusting the total number of queues which by default would be the number of logical threads allowed by the TBB scheduler. Surprisingly, this solution did not result in any change in performance and was subsequently discarded due to the added complexity.

Concurrent Vectors The final solution, which is used to produce the results in the next section, was to revert back to using containers that are the most efficient

as long as rules outlined in the Data Structures section are followed i.e. contiguous arrays. The technique utilizes three concurrent vector containers in total and two atomic variables and requires the elements that are memory managed to keep track of their index. This requirement limits the use of the manager to elements whose implementation can be changed. This issue is addressed later in this paper.

Two of the three vectors (vecObjects and vecUsed) contain pointers to elements, one for all the elements regardless of their status and the other for elements that are currently in use. The third is an (unsigned) integer container (vecAvailable) which keeps track of all the available elements by their indices. The index obtained from the array can be used to retrieve available elements. Lastly, all elements, when initialized, have an index corresponding directly to the vecObjects container.

When a request for the next available element is made, the first available index is taken from the vecAvailable, and the corresponding element in vecObjects is returned as well as recorded in vecAvailable using the same index.

Since all elements now have an index that corresponds directly to the two vector containers, recycling the elements becomes a trivial and very fast operation. When a request is made to recycle an element, its index is used to set the pointer in the in-use array to NULL and the index pushed back into the array keeping track of all the free elements.

This solution was the fastest of all the solutions and faster than the original program which used raw arrays and on the fly memory allocation/de-allocation. Although it does take more memory than the previously discussed solution because of the vector containers, the memory used remains very stable. This is the only solution that is able to morph two implicit surfaces continuously without running out of memory and is also able to perform multiple calculations with grids as large as 512^3 without running out of memory.

D.0.3 Results

The results of the original multi-threaded implementation without the memory manager were very promising in terms of performance. The downfall, as mentioned, was application stability. For the results in Figure 5.1, a quad-core Intel

Q9550 was used with 8GB of 800MHz dual-channel RAM. The surface constructed in the tests is a sphere with 80% coverage[1].

In addition to the above mentioned tests, the system was stress tested and frame rates were recorded of two surfaces morphing. The surfaces selected were the implicitly generated sphere, which is converted to a distance field in real-time and a cube mesh converted to a distance field at run-time before the start of the program. The results of morphing are show in Figure D.3. Note that the memory usage shown is taken after running the morphing stress test for 30 minutes.

The results show morphing of two distance fields at grid resolution of 128^3, 64^3 and 32^3 with a sphere of 60% coverage and a cube of 80% coverage. The highest resolution grid continuously morphs the surfaces at 9 frames per second, which is considered interactive. The 64^3 grid shows real-time results at 40 frames per second. Higher grid sizes and more complex, highly curved surfaces will likely drop the frame rate to non-interactive levels.

A major time consuming operation is the mesh batches. The application designates one mesh per core; thus, a quad-core processor will each process $\frac{1}{4}$ of the full surface. Unfortunately, Opengl or DirectX API calls to a single graphics context must be made from a single thread. Thus, in this case, only the vertex and index buffers are prepared in multiple threads but the actual draw calls are made from a single thread and the application suffers from a significant performance penalty. Mesh generation can be offloaded and done entirely on the GPU as show in [82].

If one goes through all the steps and removes the draw calls, a significant increase in FPS[2] can be observed. Furthermore, morph shapes at a higher grid resolution of 256^3 (16.7 million voxels) is possible with an average of 5.1 FPS (see Table 5.1).

[1] It is important to note the surface coverage. Higher surface coverage decreases the sparsity of a spatial data structure and causes the application to use more memory

[2] Frames per second

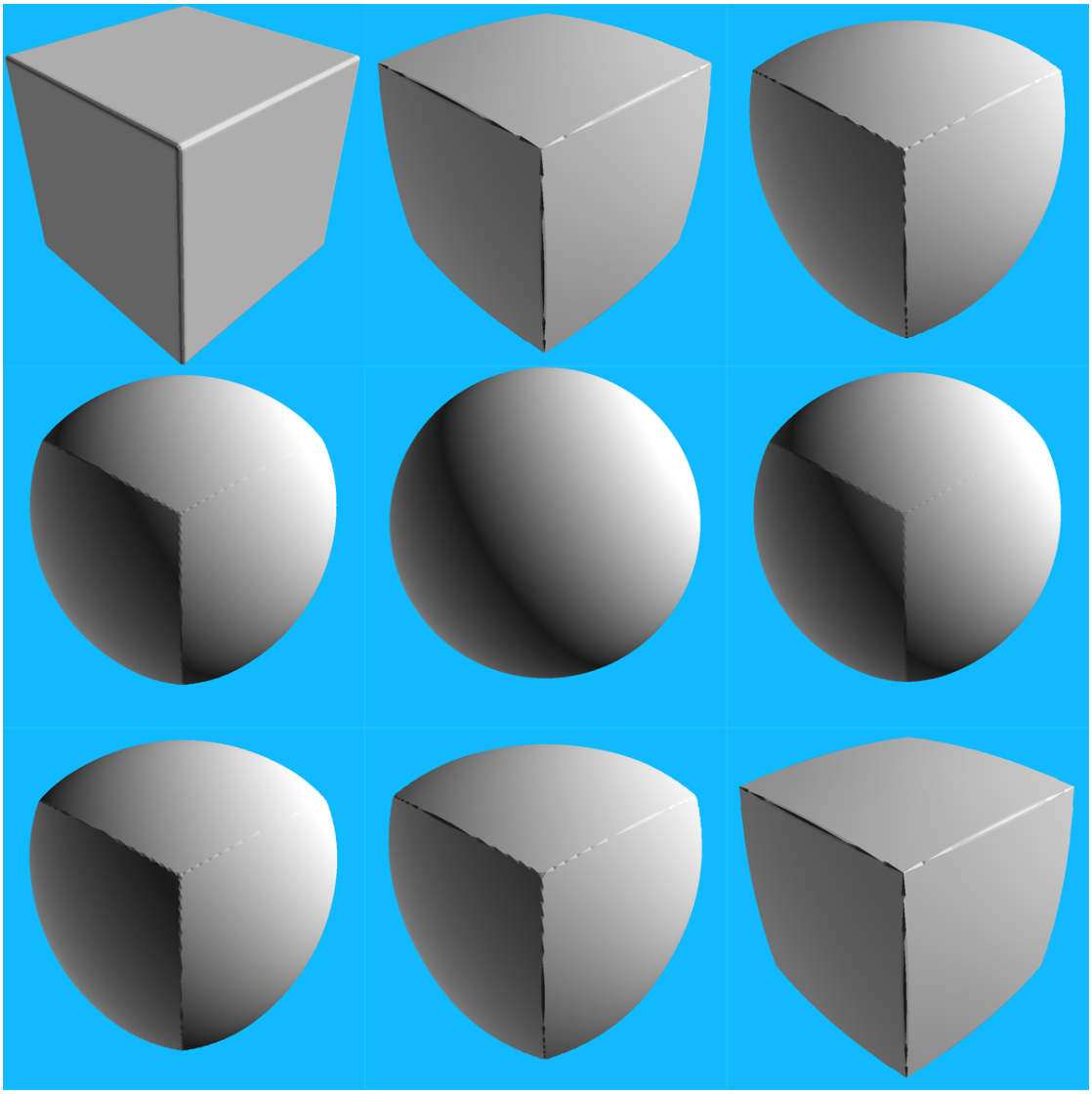


Figure D.2: A full morph cycle between two distance fields (sphere and cube) at a resolution of 256^3. The normals were not smoothed in this implementation. These screenshots were taken on a computer with an Intel Core i7 @ 2.2GHz. An average of six frames per second was maintained with a maximum memory usage of 656.25 megabytes. The memory consumption remained stable for the duration of the morph cycles.

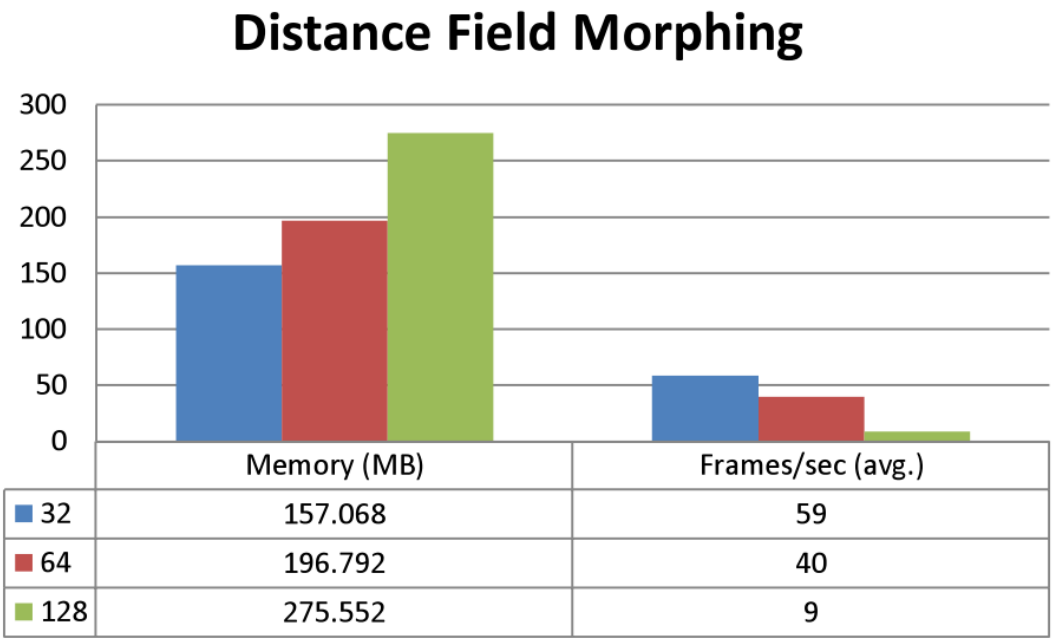


Figure D.3: Continuous morphing of two distance fields at different grid resolutions and their memory usage. The grid sizes tested were 32^3, 64^3 and 128^3. Each test was left running for 30 minutes with the memory usage monitored to ensure that the application was stable.

Appendix E

Advection (Problems & Solutions)

While implementing an efficient LSM framework capable of deforming implicit surfaces in real-time a few obstacles and *gotchas* where the logical errors were subtle enough to elude scrutiny were encountered; their (negative) impact on the stability of the distance field was significant. For details on the theory, see Section 3.3.

E.1 Deformation and Δt

The levelset framework presented in this thesis is capable of deforming surfaces based on an unordered set of points and tools that act like brushes. All deformations depend heavily on the Euler integration (equation 4.3) where the previous distance, the goal and a time step are used to calculate the final distance.

The deformation of the surface must be performed such that the fastest moving interface dictates the maximum Δt. In other words, the voxel with the largest distance to its goal must calculate the *global* Δt for the entire deformation in one step. The next step will require another calculation of Δt.

Since Euler integration is essentially *predicting* the new signed distance with each time step, selecting a very large Δt will cause the interface contours to not retain a proper gradient. In the worst case, very sharp gradients will result in a discontinuous distance field and will tear the resulting mesh. Selecting a very low Δt has no observable ill effects; the convergence in this case may take a very long time.

E.2 Calculation of Surface Normals

Marching cubes or marching tetrahedron algorithms were used to generate a mesh from a signed distance field. The resulting mesh is a polygon soup where it is impossible to generate smooth normals without another pass on the mesh. Normals can be calculated per voxel but that gives a hard faceted look to the resulting mesh (see Figure 3.9).

In this implementation a second pass for the calculation of smooth normals while generating the mesh was avoided. The solution involves calculating the normals at the same time marching cubes (or marching tetrahedron) produces the triangles by using the same tri-linear interpolation which the marching algorithms use to generate interpolated vertices for their triangles.

To start, normals for each corner of the voxel are calculated (see Algorithm 2). During a march, when a vertex is being calculated by interpolation, the respective corners of the voxel are also interpolated to calculate a normal for the interpolated vertex.

Algorithm 2 Calculate normals for each vertex of a triangle from a marching algorithm

```
for all corners of voxel do
    get perpendicular neighboring corners
    for all neighboring corners do
        calculate gradient per axis
    end for
end for
```

Appendix F

Robot with Kinect Mount

As discussed in Chapter 6 the aim was to use the custom built robot with the Kinect sensor mounted on top. Since the data from the Kinect at the height of the mount was unavailable, a tripod setup was used instead. Nevertheless, the following figures show the robot with the Kinect mounted which will be used in future experiments. Note that for the future experiments, the Kinect camera will be mounted higher preventing the laptop from occluding the IR markers and also allowing the Kinect camera to return complete depth data.

Figure F.1: The Kinect camera mounted on the robot with the help of an acrylic mount. The camera has 3 IR markers attached to it which is required by the motion capture system to compute a transformation matrix.

Figure F.2: The robot inside the test area. The area covered by black mats denote (roughly) the capture volume. The motion capture cameras, seen on the tripods, have a maximum range of 36 feet.

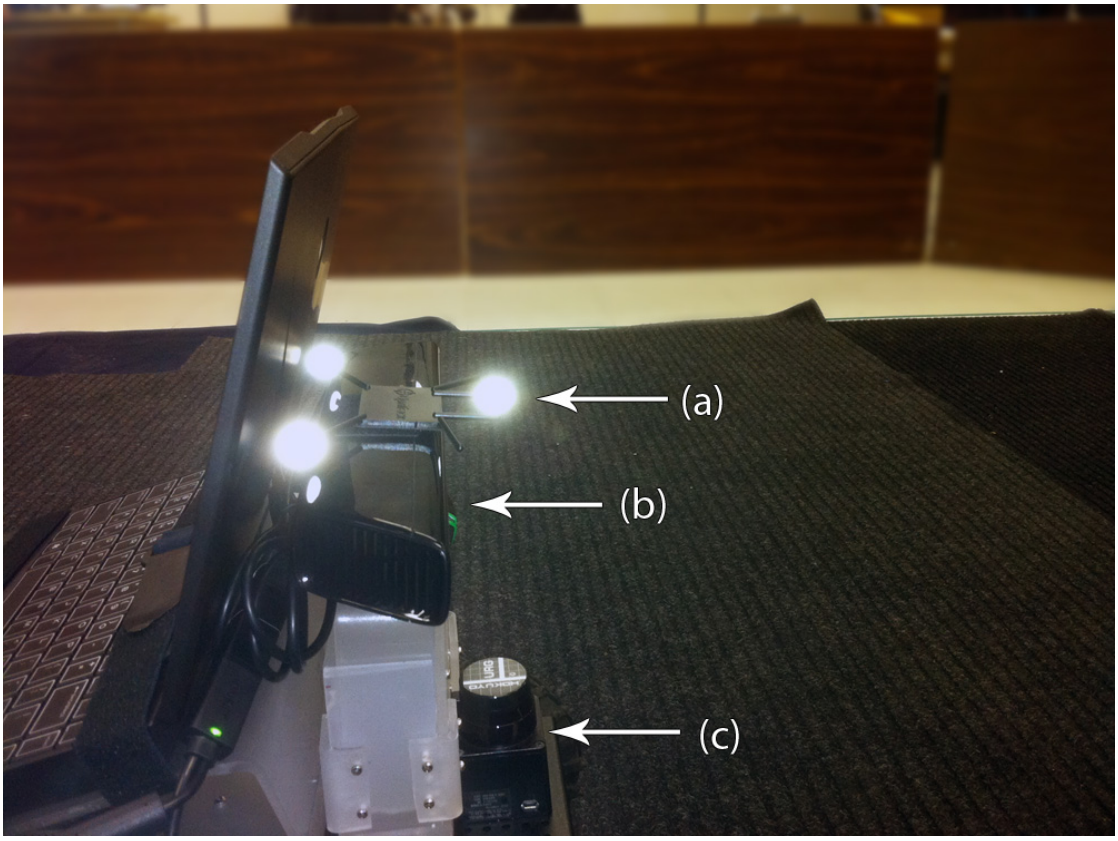


Figure F.3: (a) an IR marker for the cameras to track (b) the Kinect depth and color sensor (c) the Laser sensor used for reading in the 2D LSM framework (see Section 4).

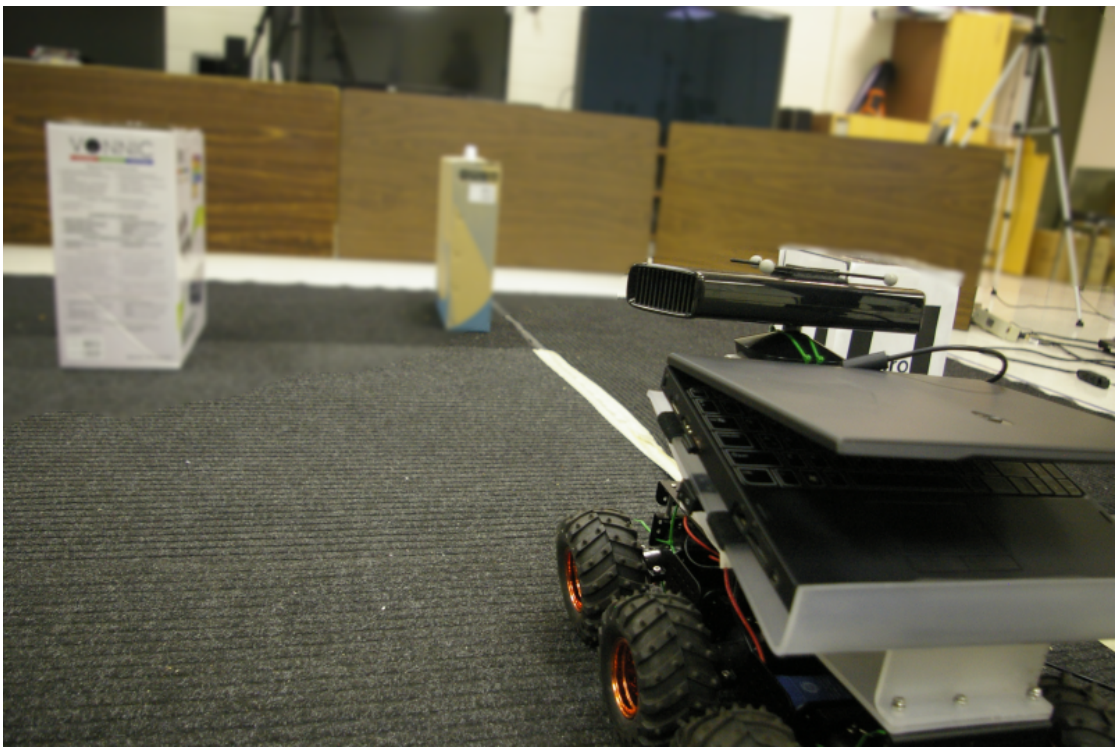


Figure F.4: The robot in the test area with simple obstacles and a perimeter wall (upturned tables). Since the Kinect camera is mounted too low, most of the floor is not captured by the camera with the result that the obstacles appear to be floating.

Appendix G

Motion Capture Calibration

To calibrate the motion capture system one must first ensure that all cameras have a clear view of the test area. This can be done in two ways:

- Switch to gray-scale mode
- Place IR markers around the perimeter

Although switching to gray-scale mode sounds better, the images from the IR cameras are poorly lit. For the experiments, the second option was used.

The calibration of the system allows the motion capture software to automatically calculate the camera positions and the capture volume. This is done with the help of a wand with an IR marker attached to the tip (Figure G.1). This marker is larger than the other markers and generally more reflective. The wand is moved in an approximately circular motion in the desired capture volume, making sure that all the cameras can see the IR marker at some point. The software shows the tracking trail of the IR marker for each camera. The goal is to fill each camera's view-able area with the trail.

Once the calibration is complete, the ground plane must be setup. This is done with a steel square with IR markers attached to the top of the frame (Figure G.2). The square is placed at a desired location, which now becomes the origin. The calibration is finally complete.

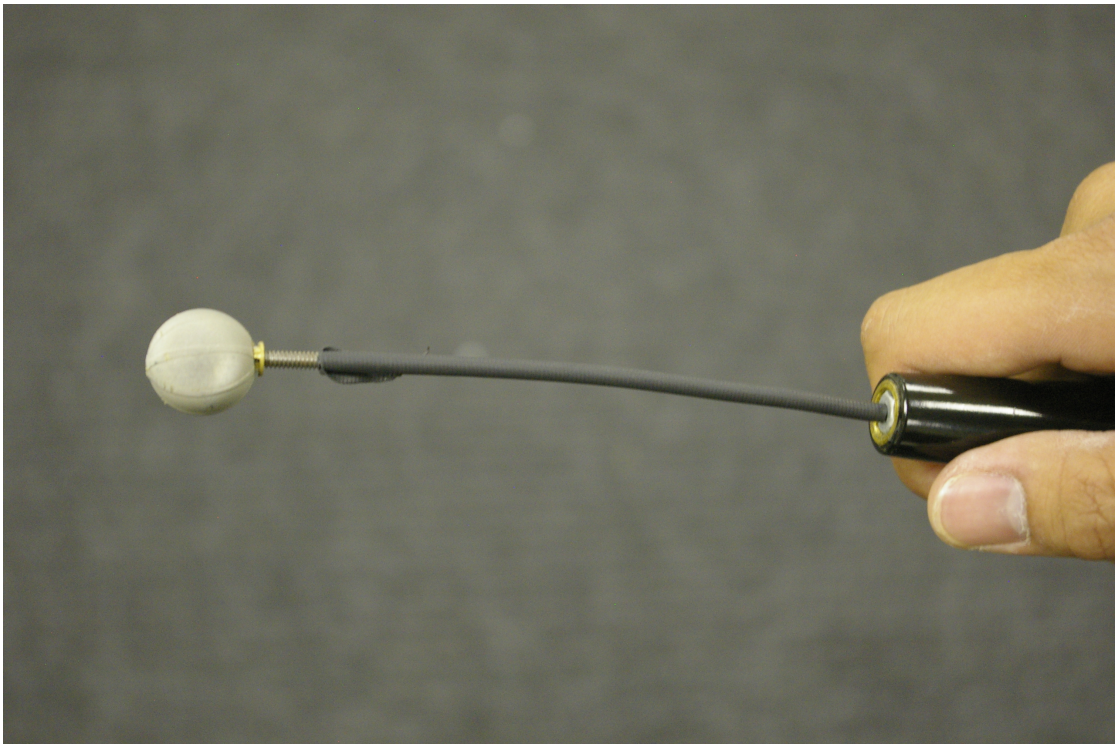


Figure G.1: This is the IR wand. It is used to calibrate the motion capture system before collecting any useful data.

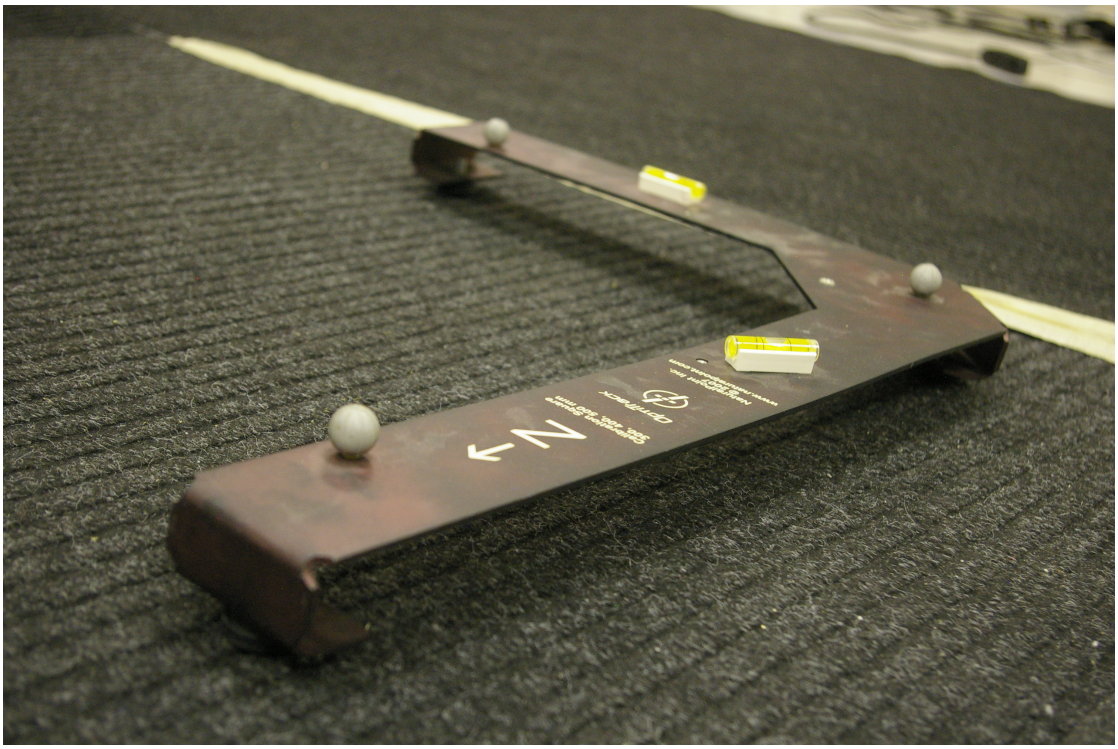


Figure G.2: A set-square with IR markers attached. This is to help setup the ground plane once the cameras have been calibrated with the IR wand.

Appendix H

Projection Formulas

For projection and deprojection of points to and from various graphical spaces the following formulas were used. Note that the equations assume that the co-ordinate system is right-handed and that the matrices are written in column major format. The formulas and derivations were graciously provided by a colleague, Daniel Buckstein. Table H.1 lists the terms used in the later equations for reference.

P_{ij}	projection matrix value at column i and row j
r, l	right and left clipping planes
t, b	top and bottom clipping planes
n, f	near and far clipping planes
eye	refers to a co-ordinate in the viewer/camera's local space
$clip$	refers to a co-ordinate in clip space, which is essentially the volume that determines whether or not a point is visible
NDC	refers to a co-ordinate within the range [-1, 1], where anything beyond this range is not rendered (shorthand for Normalized Device Co-ordinates)

Table H.1: Terms used in this appendix

Depending on the type of rendering required, two different projection matrices can be used, namely orthographic and perspective projection. The distinguishing visual characteristic between the two projection types is the way parallel lines are rendered. Two parallel lines moving away from the camera appear parallel in an orthographic projection but appear to converge in the perspective projection.

A perspective projection is written as (taken from [86]):

$$Proj_{pers} = \begin{pmatrix} \frac{2 \cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} P_{00} & 0 & P_{20} & 0 \\ 0 & P_{11} & P_{21} & 0 \\ 0 & 0 & P_{22} & P_{32} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{H.1}$$

and an orthographic projection as

$$Proj_{pers} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} P_{00} & 0 & 0 & P_{30} \\ 0 & P_{11} & 0 & P_{31} \\ 0 & 0 & P_{22} & P_{32} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{H.2}$$

Using the above projection matrices, the (x, y, z) co-ordinates can be rewritten in terms of *eye*, *clip* and NDC for conversions between different spaces.

The conversions for perspective projection are:

$$
\begin{aligned}
x_{clip} &= P_{00}(x_{eye}) + P_{20}(z_{eye}) \\
x_{NDC} &= -\frac{x_{clip}}{z_{eye}} \\
&= -\frac{P_{00}(x_{eye})}{z_{eye}} - P_{20} \\
x_{eye} &= \frac{x_{clip} - P_{20}(z_{eye})}{P_{00}} \\
&= -\frac{z_{eye}}{P_{00}}(x_{NDC} + P_{20})
\end{aligned}
$$

$$
\begin{aligned}
y_{clip} &= P_{11}(y_{eye}) + P_{21}(z_{eye}) \\
y_{NDC} &= -\frac{y_{clip}}{z_{eye}} \\
&= -\frac{P_{11}(y_{eye}}{z_{eye}} - P_{21} \\
y_{eye} &= \frac{y_{clip} - P_{21}(z_{eye})}{P_{11}} \\
&= -\frac{z_{eye}}{P_{11}}(y_{NDC} + P_{21})
\end{aligned}
$$

$$
\begin{aligned}
z_{clip} &= P_{22}(z_{eye}) + P_{32} \\
z_{NDC} &= -\frac{z_{clip}}{z_{eye}} \\
&= -(P_{22} + \frac{P_{32}}{z_{eye}} \\
z_{eye} &= \frac{z_{clip} - P_{32}(z_{eye})}{P_{22}} \\
&= -\frac{z_{eye}}{P_{32}}(z_{NDC} + P_{22})
\end{aligned}
$$

Similarly, for orthographic projections:

$$
\begin{aligned}
x_{clip} &= x_{NDC} \\
&= P_{00}(x_{eye}) + P_{30} \\
x_{eye} &= \frac{x_{clip} - P_{30}}{P_{00}}
\end{aligned}
$$

$$
\begin{aligned}
y_{clip} &= y_{NDC} \\
&= P_{11}(y_{eye}) + P_{31} \\
y_{eye} &= \frac{y_{clip} - P_{31}}{P_{11}}
\end{aligned}
$$

$$
\begin{aligned}
z_{clip} &= z_{NDC} \\
&= P_{22}(z_{eye}) + P_{32} \\
z_{eye} &= \frac{z_{clip} - P_{32}}{P_{22}}
\end{aligned}
$$

The *clip*, NDC and *eye* for z can also be calculated. For perspective projection:

$$
\begin{aligned}
z_{clip} &= -\frac{f + n \cdot z_{eye} + 2 \cdot f \cdot n}{f - n} \\
z_{NDC} &= \frac{f + n}{f - n} + \frac{2 \cdot f \cdot n}{(f - n) \cdot z_{eye}} \\
z_{eye} &= -\frac{(f - n) \cdot z_{clip} + 2 \cdot f \cdot n}{f + n} \\
&= \frac{2 \cdot f \cdot n}{(f - n) \cdot z_{NDC} - (f + N)}
\end{aligned}
$$

and for orthographic projection:

$$
\begin{aligned}
z_{clip} &= z_{NDC} \\
&= -\frac{2 \cdot z_{eye} + (f + n)}{f - n} \\
z_{eye} &= -\frac{(f - n) \cdot z_{clip} + (f + n)}{2}
\end{aligned}
$$

H.0.1 Conversion Between Different FOVs

The following equations use basic trigonometry for calculating the *right* and *top* extents of the view frustum of the camera

$$right = near \cdot tan(\frac{FOV_H}{2}) \tag{H.3}$$

$$top = near \cdot tan(\frac{FOV_V}{2}) \tag{H.4}$$

where FOV_H and FOV_V is the horizontal and vertical field of view (FOV) respectively. Similarly:

$$right = a \cdot top \tag{H.5}$$

where a is the aspect ratio ($\frac{width}{height}$). Solving for H (or V, depending on what is given):

$$H = 2 \cdot tan^{-1}(a \cdot tan(\frac{V}{2})) \tag{H.6}$$